

图灵程序设计丛书

TURING

大规模、高性能、不间断网络服务的搭建和管理

24小时365天不间断服务

服务器/基础设施核心技术



PXE・Linux・LVS/IPVS・Puppet
Nagios・Ganglia・daemontools
Apache・memcached・Squid
MySQL・DRBD・VLAN



伊藤直也 胜见祐己
[日] 田中慎司 广濑正明 著
安井真伸 横川和哉
张毅 译

人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：24小时365天不间断服务：服务器/基础设施核心技术

作者：［日］伊藤直也　胜见祐己　田中慎司　广濑正明　安井真伸　横川和哉

译者：张毅

ISBN：978-7-115-38024-1

版权声明

译者序

很多人都认为网络运维是个苦差事。的确，干这行不仅要有广而全的专业知识沉淀，还需要在面对各种突发情况时做到有条不紊地沉着应对。如此，经验就成为了相关从业人员的制胜法宝。

当我第一次翻开这本书时，我异常兴奋。这本书可以说是实实在在的经验谈，虽说这些经验并非十分“前沿”和“先进”，但这些技术在实际生产环境非常接地气。此类经验谈未曾在国内成书出版，只是只言片语地零散分布在网上。可以说，这本书的出版不仅让读者看到了作者多年经验积累的结晶，而且这些经精心编排、实实在在的“干货”更有助读者付诸实践。

从编排上看，我不得不佩服原书作者和编辑的巧妙心思。这本书不但能够围绕实际运维的需要，还能从设备 / 环境搭建、性能优化、高效管理以及对未来的展望，成体系地归纳出重点知识。书中丰富的各类技 巧及案例，不仅对于初学者及有经验的网络架构师难能可贵，而且也会给网络架构师带来莫大的启发。

本书还就架构设计方面，针对常见流程提出了独到的见解。例如本书坚持使用开源软件，以方便管理为主旨来定义工具，以完善的故障保护机制来应对各种灾害，重视性能与成本之间的平衡等。因此，本书不仅单纯满足于时刻不让服务中断，而且也深刻阐释了后端程序员这个角色的价值观。

在本书的翻译中，我得到了很多朋友，特别是图灵编辑的帮助和支持，在此表示深深致谢。但限于我自身能力有限，书中难免有错误及疏漏，如果读者发现了什么问题，或者有什么见解、技术想要交流，欢迎访问图灵社区搜索本书的名称，提交勘误，或者发 E-mail（Philip.Z@foxmail.com）与我联系。

张毅

2014 年于西安

关于本书

当今社会，社交软件、博客、购物网站等丰富多彩的网络服务充斥着我们的生活，E-Mail 和聊天工具更是大家常用的交流手段。可以说，互联网已经成为了我们生活中不可缺少的一部分。笔者也不例外，每天都在使用网络。更确切地说，无论公事还是私事，几乎每天都沉浸在网络中。

然而，从笔者的角度来看，与享受网络服务的终端“用户”相对应的就是服务的“提供者”。没错，笔者的工作就是网络、服务器的搭建和运营管理。

10 年前，说到网络和服务器，第一反应就是这是价格昂贵的设备，应该不是一个能够简单进入的领域。但是近些年来，随着在 PC 机上运行的 Linux、FreeBSD 等类 UNIX 操作系统的普及，以及硬件价格的降低、网络的普及，也让大家纷纷在家中建起了服务器。

这样的情况有助于我们随时获取基础设施的相关信息。特别是在部署方式和 Apache 守护程序的配置等操作方法方面，近些年的进步可谓惊人。对基础设施的新手工程师来说，这真是个方便的时代。

然而另一方面，在高效运营管理的实现、服务的冗余及可扩展等技术上的信息和技巧还远远不够。

就笔者而言，从搭建和运营数台、数十台乃至数百台的服务器系统来说，其中最大的困难就是缺少冗余和可扩展方面的信息。当时笔者还没有冗余和可扩展的相关知识和经验，完全不知该如何下手。而且想到为了实现这些还不得不使用昂贵的商用产品，连一些小小的实验也没能尝试。

现在回想起来，当时的想法是不对的。事实上，运用开源软件和常用的设备，即可搭建兼有冗余性和可扩展性的系统。但我们回过头来看看，当时迟迟无法下手的原因究竟是什么呢？难道不是单纯地因为“不知道有这个东西”“不知道可以这样”吗？

而这就是本书的写作动机。也就是说，本书的写作目标就是为读者搭建

兼具冗余和可扩展性的基础设施提供启示。

本书的内容是使用开源软件的 Hatena 公司和 Klab 公司的工程师团队的经验总结，与实际运作的系统密切相关。这些信息都具有实践意义，而非夸夸其谈。系统是一个体系，是由各个要素相关联构成的。本书中不仅对每个要素的技术都进行了详细的说明，还重点介绍了各个技术要素之间的关联。但是本书并不是一本技术手册，所以并没有逐步对安装顺序进行说明，而且也不是说按照书中的命令去运行就一定能得到什么。

本书记述的是笔者在实际的开发现场所进行的思考、所面临的问题，以及为解决问题所做的努力和成果。希望在读者接下来设计、搭建和运营基础设施时，本书的内容能够为你提供参考。

作者代表　广濑正明

本书概述

本书由 6 章组成。

第 **1** 章　服务器及基础设施搭建入门**……**冗余及负载分流的基础

第 **2** 章　优化服务器及基础设施的拓扑结构**……**冗余、负载分流、高性能的实现

第 **3** 章　进一步完善不间断的基础设施**……DNS** 服务器、存储服务器、网络

1 ～ 3 章的主题是如何设计兼具冗余性以及可扩展性的基础设施。

每一章都是相互独立的，但是在“从较小的系统出发来搭建基础设施”这一大流程中，它们又是相互关联的。建议首先通读 1 ～ 3 章以把握整个流程，然后再回过头来细读感兴趣的章节。

第 **4** 章　性能优化、调整**……Linux** 单个主机、**Apache**、**MySQL**

第 4 章的主题是提升性能。

通过服务器的负载均衡来提升整个系统的性能。在这一过程中，单个服务器的性能优化是不可或缺的。第 4 章将针对单个服务器的性能可能遭遇的瓶颈，对其界定及优化方法进行介绍。

第 **5** 章　高效运行**……**确保服务的稳定提供

第 5 章的主题是监控和管理。

随着服务器数量的增加，运营成本也不断加大，那么运营成本将会成为瓶颈，进而就可能导致不能像期望中那样扩展基础设施了。换句话说，通过各种办法在最大程度上提高运行效率，是搭建具备可扩展性的基础设施的关键。第 5 章将以笔者身边的生产环境为例，来说明如何提升设备的运行效率。

第 6 章　服务后台……自律的基础设施、稳健的系统

第 6 章将对 Hatena 公司与 KLab 公司的 DSAS 实际运作的网络和服务器基础设施进行介绍。

笔者作为基础设施团队的领头人物，除了一些技术性的内容之外，还加入了前面章节中未能介绍的细节、至今为止的发展历程，以及自己作为基础设施工程师的动机和心理等非常有趣的内容，可读性很强。

章节作者一览表及出处

章节	作者
1.1 冗余的基础	安井 真伸（KLab）
1.2 实现 Web 服务器的冗余……DNS 轮询	安井 真伸
1.3 实现 Web 服务器的冗余……通过 IPVS 进行负载均衡	安井 真伸
1.4 路由器及负载均衡的冗余	安井 真伸
2.1 引入反向代理……Apache 模块	伊藤 直也（Hatena）
2.2 增设缓存服务器……Squid、memcached	伊藤 直也
2.3 MySQL 同步……发生故障时的快速恢复[1]	广濑 正明（KLab）
2.4 MySQL 的 Slave + 内部负载均衡器的灵活应用示例[2]	广濑 正明
2.5 选择轻量高速的存储服务器	安井 真伸
3.1 DNS 服务器的冗余	安井 真伸
3.2 存储服务器的冗余……使用 DRBD 实现镜像	安井 真伸
3.3 网络的冗余……驱动绑定、RSTP	胜见 祐己（KLab）

3.4 引入 VLAN……使网络更加	横川 和哉（KLab）
4.1 基于 Linux 的单个主机的负载评估	伊藤 直也
4.2 Apache 的优化	伊藤 直也
4.3 MySQL 的调优诀窍[3]	广濑 正明
5.1 服务状态监控……Nagios	田中 慎司（Hatena）
5.2 服务器资源的监控……Ganglia [4]	广濑 正明
5.3 高效的服务器管理……Puppert	田中 慎司
5.4 守护进程的工作管理……Daemontools	广濑 正明
5.5 网络引导的应用……PXE、initramfs	胜见 祐己
5.6 远程维护……维护线路、Serial Console、IPMI	胜见 祐己
5.7 Web 服务器的日志处理……syslog、syslog-ng、cron、rotatelogs	胜见 祐己
6.1 Hatena 网站的内容	田中 慎司
6.2 DSAS 的内容	田中 慎司

[1] 《WEB+DB PRESS》（Vol.22）特辑 2“MySQL 配置指导”、第 2 章“生产环境中的同步详解”

[2] 《WEB+DB PRESS》（Vol.38）连载“快看！这是高手的诀窍”可扩展的 Web 系统工房“第 1 回：各种各样的负载均衡”

[3] “5 分钟完成 MySQL 的内存关系调整！” ***URL*** http://dsas.blog.klab.org/archives/50860867.html

[4] 《WEB+DB PRESS》（Vol.40）连载“快看！这是高手的诀窍”可扩展的 Web 系统工房“第 3 回：监控的种种”

术语整理

从网络到应用程序，本书内容涉及范围较广，其中出现了较多的术语。首先将常用的术语整理如下。

AP 服务器（**Application Server**）

应用服务器，即能返回动态内容的服务器。

比如 Apache ＋ mod_perl 运行的 Web 服务器及 Tomcat 等应用程序运行的服务器。

CDN（**Content Delivery Network**，内容分发网络）

发送内容的网络系统。用于提高信息发送的性能和实用性。

以 Akamai 等商用服务为例，其结构上的特点是：从散布在全世界的缓存服务器中，选择离客户端较近的服务器来发送信息，据此实现性能的提升。

IPVS（**IP Virtual Server**，**IP** 虚拟服务器）

LVS（Linux Virtual Server）的成果之一，实现了负载均衡器中不可或缺的负载分流功能。

→ 参考“LVS”

LVS（**Linux Virtual Server**，**Linux** 虚拟服务器）

Linux 中旨在搭建具有可扩展性的、实用性较高的系统的项目。项目成果之一即为 Linux 负载分流所设计的 IPVS。

原先为项目名，现通常作为“基于 Linux 的负载均衡器”的意思使用。

URL http://www.linuxvirtualserver.org/

NIC（**Network Interface Card**，网络接口卡，简称网卡）

原本是指追加网络功能所需的扩展卡。有时也作为网络接口的总称使用，不区分是扩展卡还是板载。

同时也可称为 LAN 卡、网络适配器等。

Netfilter

Linux 内核中操作网络数据包所需的协议框架。

执行分组过滤的 iptables 以及实现负载均衡的 IPVS 也应用了本Netfilter 协议。

OSI 参考模型

用来描述数据通信网络层的模型，分为七层（Layer）框架。

以下为常见的层。

- 第七层（应用层）：HTTP 及 SMT 等通信协议
- 第四层（传输层）：TCP 及 UDP
- 第三层（网络层）：IP、ARP 及 ICMP
- 第二层（数据链路层）：以太网等

另外，像“L2 交换机”这样，有时也将“第 n 层”记为“Ln”。顺带一提，OSI 是 Open Systems Interconnection 的缩写。

VIP（**Virtual IP Address**，虚拟 **IP** 地址）

不同于物理性质的服务器及网卡，该 IP 地址会被浮动地分配某项服务或功能。

例如对于负载均衡器，接收客户端请求的 IP 地址就称为 VIP。这是因为该 IP 地址对 HTTP 等服务进行了关联，另外在冗余的 Active/Backup 架构中，唯一的 Master，即 Active 的负载均衡器也继承了该 IP 的行

为。

虚拟地址通常也称为虚拟 IP 地址。

可用性（**Availability**）

系统停止的可能性。在可用性较高的情况下，通常该服务不会随意终止。另外，根据其字面意思，也可理解为“运行效率高”或者“1 年中的运作时间长”等。

内容（**Contents**）

在网络服务的环境中，内容是指返回给用户浏览器的 HTML 或图片等数据。

静态内容是指不会发生变化的内容，例如 HTML 或图片等；动态内容是指会变化的数据，根据请求的不同所返回的内容也不同。在某些情况下，动态内容并非单纯指数据本身，而是指返回动态数据的服务器站点的程序。

服务器集群（**Server Farm**）

很多服务器集合而成的基础系统。根据上下文环境，有时也作为硬件设施的意思使用，与数据中心的意思相同。

在一些新闻中，有时也会形象地称为“服务器农场”。

冗余（**Redundancy**）

将系统的构成要素配置多个，这样即使其中一个因为发生故障而停止运作，也可以立即切换到备用设备以使服务不停止。

RAID（Redundant Arrays of Inexpensive Disks）是冗余的典型例子。

交换集线器（**Switching Hub**）

目前市场上几乎所有的集线器都是带有桥接功能的交换集线器，而非“中继集线器”（Repeater Hub）。

有时也称为 L2 交换机，或者简单地称为交换机。

可扩展性（**Scalability**）

随着用户的增多以及规模的扩大，在某种程度上扩展系统以加强应对的能力。

横向扩展（**Scale-out**）

通过将内容分散到多台服务器并行处理，来提升系统整体的性能。

例如使负载均衡器下配置的 Web 服务器的数量翻倍等。

纵向扩展（**Scale-up**）

通过提升单个服务器的性能，来提升系统整体的性能。

例如增加服务器内存、换代到更高性能的服务器等。

准生产环境（**Staging Environment**）

在投入真正的服务前，进行最终的动作确认的环境（→ 可参考“生产环境”）。

吞吐量（**Throughput**）

在网络等数据通信环境中使用，代表单位时间的传送量（→ 可参考“延迟”）。

例如，虽然同样是车，但和 F1 赛车相比，大巴车可乘坐的人较多，因此大巴车的“吞吐量”就较大。

单点故障（**Single Point of Failure**）

若此处出现问题，就会令整个系统停止，即系统的要害。也叫作 SPO（Single Point of Failure）。

例如，即使服务器由 RAID 和多路复用的电源构成，如果全部服务器都连接在同一台交换集线器上，从整个系统来看这台交换集线器即为单点

故障。

数据中心（**Data Center**）

为了容纳服务器设备而创建的专用设备的名称。

安装有空调，并配备停电、火灾、地震等问题的应急措施，以保证每时每刻都能够正常提供服务。

守护程序（**Daemon**）

在后台下持续运行并发挥某种作用的程序。

例如 httpd 和 bind 等。

网段（**Network Segment**）

广播数据包所及范围内的网络段。虽和“冲突域”（Colision Domain）意思相近，但因为很多情况下并无冲突发生，所以很难再说“Network Segment ＝ Colision Domain”了。

网络引导（**Network Boot**）

通过网络获取启动时必要的引导加载程序和内核映像并启动。

5.5 节介绍的 PXE 是实现网络引导的方式之一。

分组（**Packet**）

通常指 IP 中数据的最小计量单位。有时也叫 IP 分组、IP 包、数据包等。

故障转移（**Failover**）

在冗余系统中，在活动节点（Active Node）（服务区或者网络设备）停止时，自动通过某种行为切换到备用节点（Backup Node）。

顺带一提，如果不是自动切换，而是手动切换，通常叫作 Switch over（手动切换式故障转移）。

故障恢复（**Failback**）

从活动节点停止进行故障转移的状态，恢复到原始的正常状态。

帧（**Frame**）

以太网中数据的最小计量单位。也称为以太网帧（Ethernet Frame)。

被阻塞（**Blocked**）

为了等待读出或写入处理的结束而无法进行其他处理的状态，称为“因等待 I/O 而被阻塞”。

主要是针对磁盘 I/O 和网络 I/O 使用的术语，在输入输出处理时一般也会用到。

生产环境（**Production Environment**）

服务的运行环境（→ 参考“准生产环境”）。

健康检查（**Health Check**）

确认检查对象的状态是否正常。

例如确认 Web 服务器是否能够响应 ping、是否能连接 TCP 的 80 端口、是否能应答 HTTP 等。通常情况下，若健康检查失败，就会向管理者发出监控对象故障的警示信息。

有时也称为“服务存活状态的监控”。

负载（**Load**）

“负载”的种类很多，大致可分为“CPU 负载”和“I/O 负载”。

衡量负载情况的指标通常是 load average（平均负载）这样的数值。此外 vmstat 及 top 等命令也可衡量负载。具体请参见 4.1 节。

瓶颈（**Bottleneck**）

阻碍系统整体性能提升的地方。

内存文件系统（**Memory File System**）

并非像磁盘那样永久性的存储装置，而是在内存中建立的文件系统。

虽说使用起来类似磁盘上的文件系统，但由于存储在内存中，因此一旦重启数据就会丢失。但其拥有读写速度快等优点。

轮询（**Round Robin**）

对多台节点有序地派发请求。

包括 DNS 轮询和负载均衡算法等。前者是指将多个 A 记录（IP 地址）分配到一个 FQDN（完全限定域名，Fully Qualified Domain Name）上以分散请求，后者是指将请求按顺序分散到多台服务器上。

资源（**Resource**）

指 CPU、内存、磁盘等服务器的硬件资源。

通常说“资源被占据”就是指 CPU 使用率过高。

延迟（**Latency**）

在网络等数据通信领域里使用时，通常指数据投递完成所花费的时间（→ 参考“吞吐量”）。

比如说，同样是车，F1 赛车就比大巴车更快速，延迟更小。

层（**Layer**）

→ 参考“OSI 参考模型”。

负载均衡器（**Load Balancer**）

位于客户端与服务器之间，将客户端的请求分散到后端的多台服务器。

换句话说，就是将多台服务器合并为一台高性能的虚拟服务器的装置。

第 1 章　服务器及基础设施搭建入门——冗余及负载分流的基础

1.1　冗余的基础

1.1.1　冗余概述

冗余（Redundancy）是指，在故障发生时，使用事先准备好的备份设备，使系统相关功能得以继续提供服务。比如工厂或者医院为了防止停电，一般都准备有发电装置；而公共交通工具为了以防万一，也配备有多个制动系统以确保安全。

提供 Web 服务的网络与服务器系统也不例外，为了确保服务的可用性，对系统进行冗余处理的情况并不少见。本节将讲解实现系统冗余的基础知识，然后再介绍个简单的例子。

1.1.2　冗余的本质

系统的冗余可以通过以下步骤实现：

❶ 设想可能发生的故障

❷ 根据故障准备备份设备

❸ 部署故障发生时切换到备份设备的工作机制

下面按照以上各个步骤，对操作流程进行简单的介绍。

❶ 设想可能发生的故障

冗余的第一步就是设想故障发生时的情况。请参考图 1.1.1 中简单的系统拓扑结构，来考虑该图中可能会导致系统故障的原因。

首先，列出图 1.1.1 所示的系统可能发生的故障。

- 路由器故障所造成的服务停止
- 服务器故障所造成的服务停止

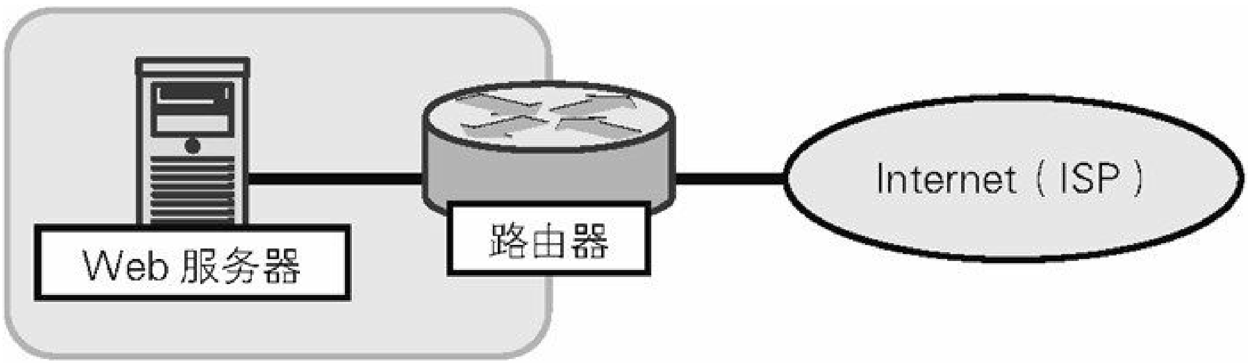


图 **1.1.1**　最简易的服务器系统

在图 1.1.1 中，以上任何一种情况发生，都会造成服务停止。

❷ 预先准备好备份设备

接下来，为了应对故障而预先准备备份设备。在图 1.1.1 的拓扑结构中增设一套备份设备，得到的拓扑结构如图 1.1.2 所示。至此，备份路由器和备份 Web 服务器还尚未连接到网络。

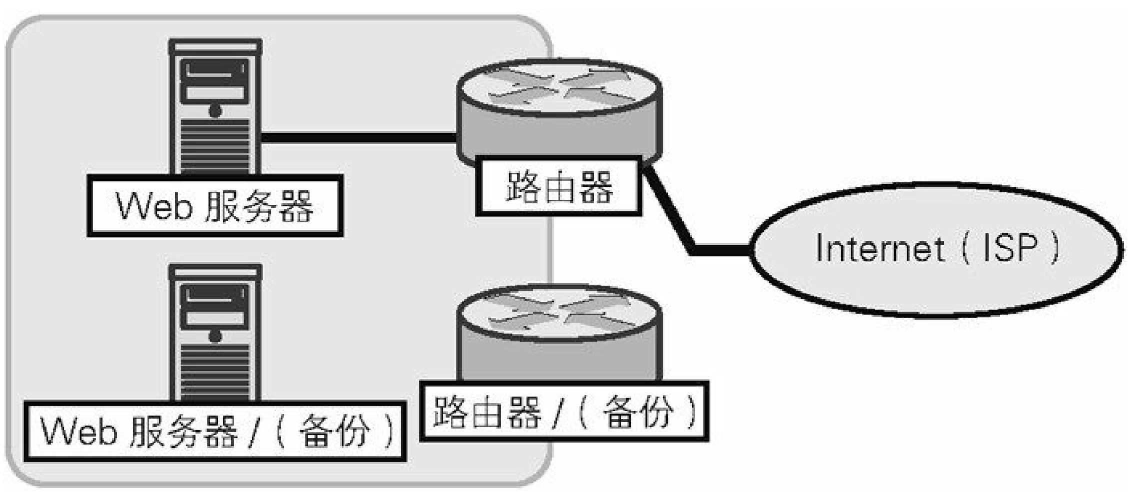


图 **1.1.2**　增设备份设备

❸ 部署工作机制 **……**当故障发生时，切换到备份设备

最后，部署工作机制。部署工作机制通常是指，针对以上 ❶ ❷ 步骤中可能发生的故障节点 [1] 及故障属性，考虑通过在硬件层面部署怎样的拓扑才能解决这些故障。

> [1] 在计算机科学技术名词 [MINGC194] 的第 112 页中，关于计算机网络的名词 “node”对应的翻译是“结点”，但我们这里将“node”视为有计算能力的单元，就像人体的关节一样，因此译为“节点”。——译者注

下面以步骤 ❶ 中所设想的路由器故障和 Web 服务器故障为例，对基本工作机制的部署和冗余中用到的基本术语进行讲解。

1.1.3　应对路由器故障的情况

在图 1.1.1 的状态下，一旦路由器发生故障，服务就会停止。在图 1.1.2 中，通过增设备份设备，即便路由器发生故障，也能像图 1.1.3 中 那样仅通过切换连接线就可以方便地解决故障，从而恢复服务。

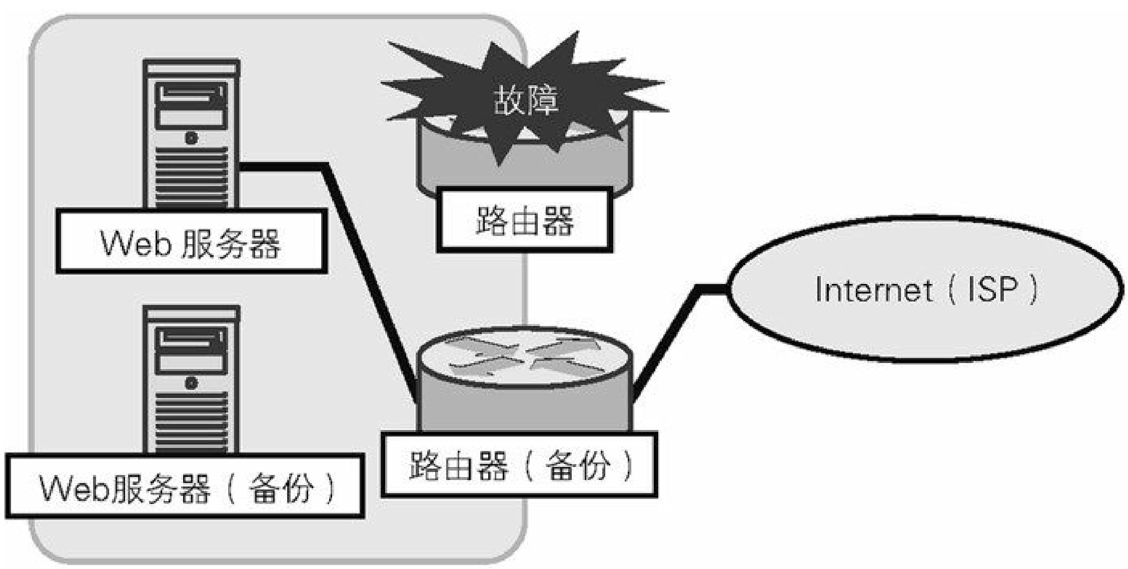


图 **1.1.3**　应对路由器故障的情况

冷备份

如图 1.1.2 和图 1.1.3 所示，我们平常并不会用到备份设备，只有在故障发生时才需要连接到备份设备，该工作机制称为“冷备份”（Cold Standby）。

在此需要关注的重点是：现用设备与备份设备的相关设定需要完全一致。即在冗余系统中"必须确保现用设备始终与备份设备的架构保持同样的状态"。

在使用路由器等网络设备的情况下，因为不用在生产环境中频繁地修改设定（切换连接），而且数据也并非一定要长期保存，所以使用冷备份不失为一个现实可行的方法。

1.1.4　应对 **Web** 服务器故障的情况

下面考虑在 Web 服务器发生故障的时候要如何应对。当 Web 服务器发生故障时，也可以采取与上述路由器发生故障时同样的思路，即参照图 1.1.4 切换到备份设备，但此处仍存在一个问题。

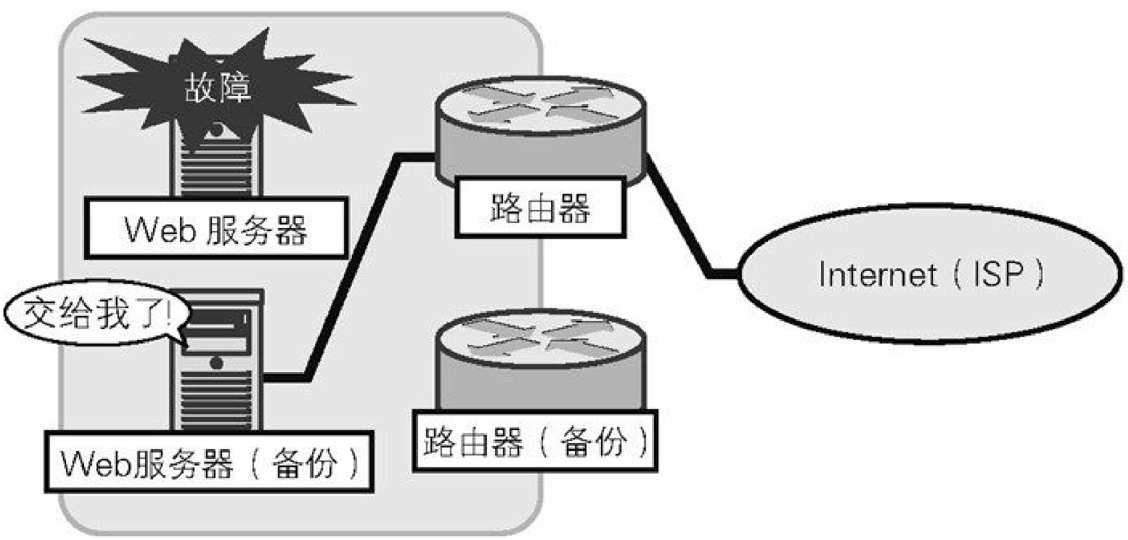


图 **1.1.4**　应对在 **Web** 服务器故障的情况

热备份

综上所述，在冗余系统中"确保现用设备始终与备份设备的架构保持同样的状态"是非常重要的。

在 Web 服务器中，面对每天都要更新的网站内容，应用软件或操作系统的版本更新等也在所难免。为了能启用备份设备，就不能停止冷备份设备在生产环境中的各种更新。仅仅为了在发生故障时能启用备份设备，而去费时费力地同步更新站点内容或应用软件版本，这确实有些费

力不讨好。

有个好办法可以解决该问题，即让 Web 服务器的备份设备一直保持接入电源及 Internet 的状态。在现用设备内容更新时，备份设备也同步更新内容以确保与现用设备的内容一致，如图 1.1.5 所示。

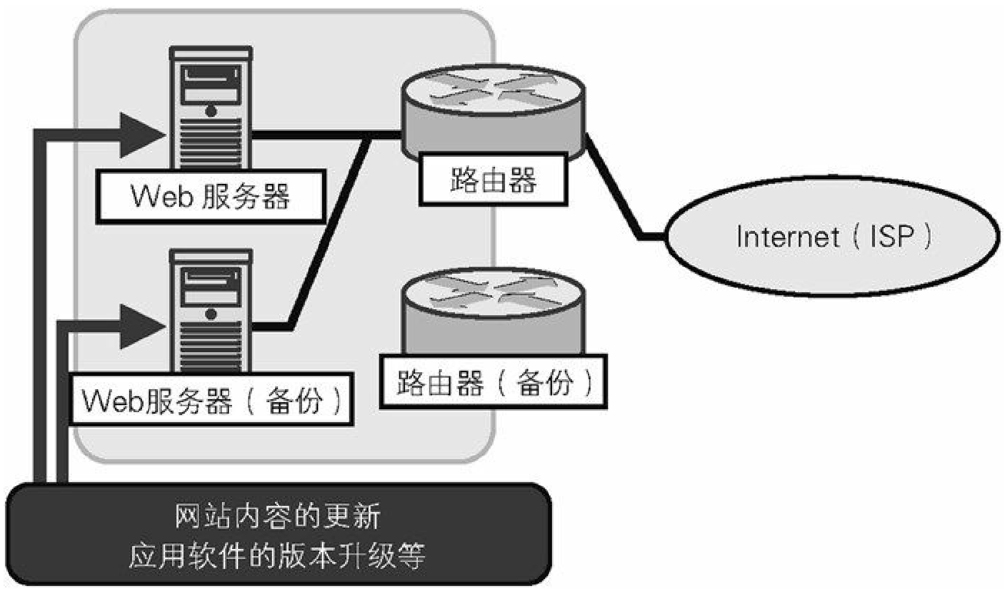


图 **1.1.5**　热备份模式的使用

像这样让两台服务器同时运行，并保持同样状态的使用模式称为“热备份”（Hot Standby）。在使用热备份的情况下，由于不会物理性地插拔网线以切换连接，发生故障也不会宕机太长时间，所以就让及时切换以解决故障成为了可能。

1.1.5　故障转移

当现用设备发生故障时，系统会自动将处理交接到备份设备，该操作称为故障转移 （Failover）。

服务器的故障转移主要使用“虚拟 IP 地址”（Virtual IP Address，下文统称 VIP）与“IP 地址的映射（也称漂移）”来操作。

VIP

图 1.1.6 是使用 VIP 搭建 Active/Backup（现用设备 / 备份设备）拓扑结

构的例子。图中的 Web1 是现用设备，目前 VIP（10.0.0.1）已经映射到该设备使用的 IP 地址，通过连接到该 VIP 来提供 Web 服务。

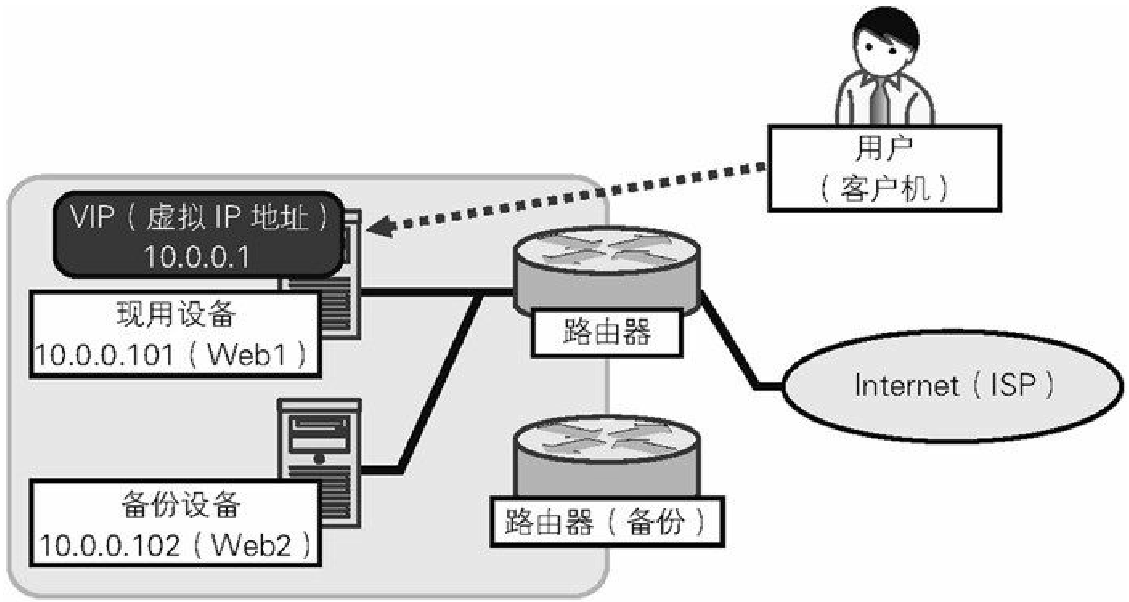


图 **1.1.6**　利用 **VIP** 搭建的 **Active/Backup** 拓扑结构

IP 地址的映射

如图 1.1.7 所示，当现用设备发生故障时，VIP 就会映射到备份设备，让最终用户访问到备份设备 Web2。

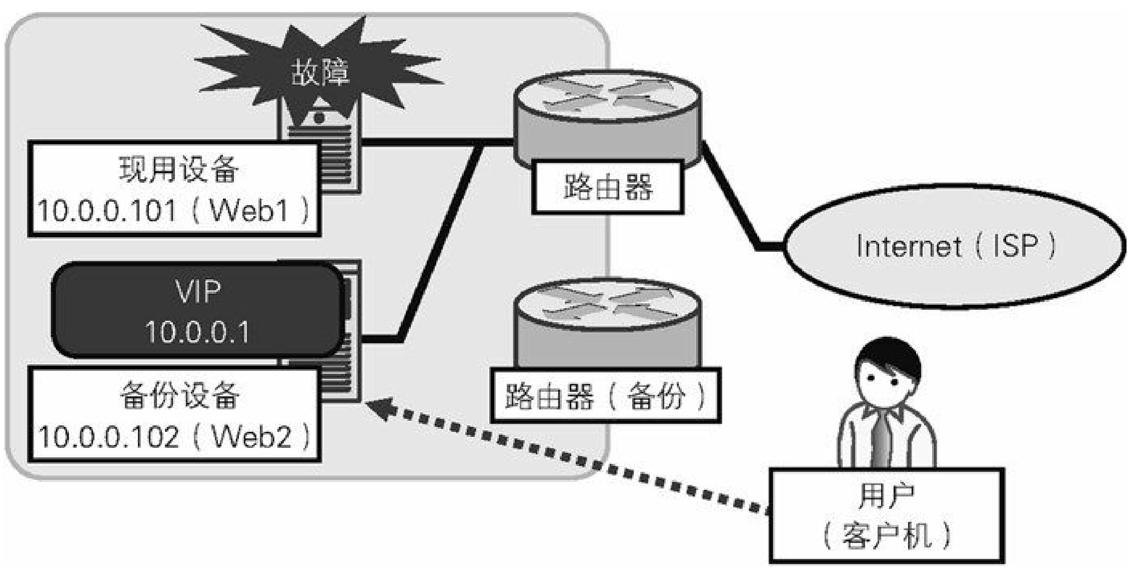


图 **1.1.7**　**IP** 地址的映射

1.1.6 检测故障……健康检查

为了能正常进行故障转移，需要在现用设备发生故障时及时知晓，该操作机制称为健康检查（Health Check）。健康检查有各种类型，可以根据用途选择适合的方法。以下举出几种常用的方法：

- **ICMP** 监控（第三层[2]）

  ICMP 监控是指，检查由 ICMP[3] 所发出的 echo 请求是否得到了应答。由于这是最基本的健康检查，因此无法得知 Web 服务（Apache 等）是否已经停止

- 端口监控（第四层）

  端口监控是指，通过使用 TCP 协议尝试连接目标主机，检查目标主机是否能被连接。该监控可以得知 Web 服务是否还在正常运行，但无从得知系统的负载状况，也无从得知错误的回报情况

- 服务监控（第七层）

  通过实际发出 HTTP 请求等，检查该请求是否得到了应答。虽说这种方法可以检查所有的异常状况，但根据情况的不同，可能会加重目标主机的负载

> [2] 这里的层次指的是 OSI 参考模型中的七层结构，下文同。——译者注

> [3] 全称为 Internet Control Message Protocol，是异常发生时通知错误及错误信息的协议。

Web 服务器的健康检查

在本章的开头，我们设想了在如图 1.1.1 所示的拓扑结构下可能会发生的两种故障。

其中一种是“服务器故障所造成的服务停止”，为了检测出该故障，需要使用上文介绍的健康检查中的“服务器监控”。为什么需要使用服务器监控呢？因为即使服务器已经接入电源，而且对 ICMP 测试有应答，我们也依然不能断定 Web 服务（Apache 等）在正常工作。为了检测出 Web 服务器的故障，需要尝试向需要测试的目标主机实际发出 HTTP 请求，

并确认能否得到应答，这是相对切实有效的方法。

路由器的健康检查

而为了检测出“路由器故障所造成的服务停止”，就需要使用 ICMP 监控了。请注意这里并不是对路由器进行 ICMP 监控。因为需要确认的是“路由器是否正确进行了分组（Packet）交换”，因此从网络上的其他主机来监控 Web 服务器是比较好的选择。总而言之，只要确定 Web 服务器与 Internet 能够进行通信就可以了。

*　　*　　*

在进行健康检查时，首先要明确目标，这一点最为重要。

1.1.7　搭建 Active/Backup 的拓扑结构

下面就实际使用 Shell 脚本，尝试搭建前文中如图 1.1.6 所示的拓扑结构。这里 Web1 与 Web2 已经分配了只属于本机的实际 IP 地址。在代码清单 1.1.1 中，每秒对 VIP 发出一次`ping` 测试，若失败，则该脚本会自行将 VIP 映射到可用主机上。

首先，请在 Web1 中执行代码清单 1.1.1 的脚本。在输出字符串“`fail over!`”后，该脚本执行结束，于是 VIP 就被分配到了 Web1。接下来，请在 Web2 中执行代码清单 1.1.1 中的脚本，这次每秒都会有一行“`health ok!`”输出。

代码清单 **1.1.1　failover.sh**

```
#!/bin/sh
VIP="10.0.0.1"
DEV="eth0"

healthcheck() {
  ping -c 1 -w 1 $VIP >/dev/null
  return $?
}

ip_takeover() {
  MAC=`ip link show $DEV | egrep -o '([0-9a-f]{2}:){5}[0-9a-f]{2}' |
head -n 1 | tr -d :`
  ip addr add $VIP/24 dev $DEV
```

```
  send_arp $VIP $MAC 255.255.255.255 ffffffffffff ←❶
}
while healthcheck; do
  echo "health ok!"
  sleep 1
done
echo "fail over!"
ip_takeover
```

※ 请在Debian/GNU Linux 4.0（内核版本 3.1）以上的版本中执行上面的代码。

请在客户端对 VIP 执行 ping 指令，并且在执行的同时尝试关闭 Web1。一旦 Web1 关闭，Web2 上运行的脚本的健康检查就会失败，从而就会有 VIP 映射的操作。

如图 1.1.8 所示，通过在客户端执行的 ping 指令所取得的结果，能够确认在 Web1 关闭大概三秒钟后，IP 地址的映射操作已经完成。

图 **1.1.8**　故障转移的动作确认

```
~$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=2.46 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=1.86 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=5.06 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=2.64 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.453 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=3.73 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=3.91 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=0.418 ms ←在这里关闭Web1
64 bytes from 10.0.0.1: icmp_seq=11 ttl=64 time=3.20 ms ←已经完成Web2的交接
64 bytes from 10.0.0.1: icmp_seq=12 ttl=64 time=1.69 ms
64 bytes from 10.0.0.1: icmp_seq=13 ttl=64 time=1.48 ms
```

IP 地址的映射操作

“IP 地址的映射操作”并不是单纯的“仅仅更换 IP 地址”这么简单。为了

对此加以验证，我们尝试为两台服务器分配同一 IP 地址，从其他设备持续发出`ping` 指令，并交替拔插 LAN 网线。可以发现，无论怎么拔插网线，通过`ping` 指令也只能确认单台服务器的状态。

在 LAN（Ethernet，以太网）中，并不是通过 IP 地址，而是通过被固定分配给 NIC（Network Interface Card，网络适配器）的 MAC（Media Access Control Address，介质访问控制）地址来完成通信的。在给其他服务器发送分组的时候，为了取得 MAC 地址，通常还会使用到 ARP（Address Resolution Protocol，地址解析协议）。

ARP 协议是指，通过指定目标设备的 IP 地址，查询目标设备的 MAC 地址。但由于每次通信时都进行查询效率会非常低，因此通常会将一次取得的 MAC 地址缓存在 ARP 缓存表中一段时间。这样一来，即便别的服务器被分配到了相同的 IP 地址，在 ARP 缓存表更新之前，该服务器也都没有办法完成通信。所以在分配 IP 地址到其他服务器时，请务必注意更新 ARP 缓存表。

你也可以使用 **gratuitous ARP** （GARP）作为获取 MAC 地址的手段。在通常没有使用 gratuitous ARP 的情况下要查询 MAC 地址时，ARP 请求的内容是“请告诉我这个 IP 地址所对应的 MAC 地址”。而若是使用了 gratuitous ARP，gratuitous ARP 则会通知其他主机：“这是我的 IP 地址和 MAC 地址。”在代码清单 1.1.1（failover.sh）中，在 ❶ 处使用 了`send_arp` 命令，目的就是发送出 gratuitous ARP 的通知消息。

1.1.8　还想更有效地使用服务器 ……负载分发 [4]

> [4] Load Balance，文中在强调分流的逻辑动作时译为了“负载分发”；强调负载的拓扑时译为了“负载均衡”；强调降低负载时译为了“负载分流”。——译者注

在上述的 Active/Backup 拓扑结构中，仅使用了现用设备进行处理，而备份设备则闲置没用，仔细想想还真是可惜。若是能同时使用两台服务器来提供服务，那网站整体的处理性能应该会翻倍吧。

使用多台服务器分发处理，网站整体的可扩展性也会随之提高，这样的做法称为被负载分发 （Load Balance，或负载均衡）。对 Web 服务器做出负载分发的处理后，将来即便访问量变高，目前的服务器已经无法满足需求，也可以轻易地通过增设服务器来满足相应的需要，而并不需要购买高性能服务器来更换掉现有的无法满足需求的设备。这样一来，即便是旧的服务器也可以贡献余热，也就不会造成浪费。

在下面的 1.2 节和 1.3 节，将介绍 Web 服务器负载分发的具体搭建实例。

1.2　实现 **Web** 服务器的冗余 **——DNS** 轮询

1.2.1　**DNS** 轮询

DNS 轮询 （DNS Round Robin）是指，利用 DNS 把一台服务器需要处理的内容，分配到多台服务器来进行。如图 1.2.1 所示为 DNS 轮询的行为。假定有两名想要访问 www.example.cn 网站的用户：A 先生与 B 先生，这两人分别向 DNS 服务器询问了 www.example.cn 网站的 IP 地址，其后 DNS 服务器分别向两人解析返回了“x.x.x.1”与“x.x.x.2”两个不同的 IP 地址，于是 A 先生便通过 x.x.x.1 访问，B 先生则通过 x.x.x.2 访问。

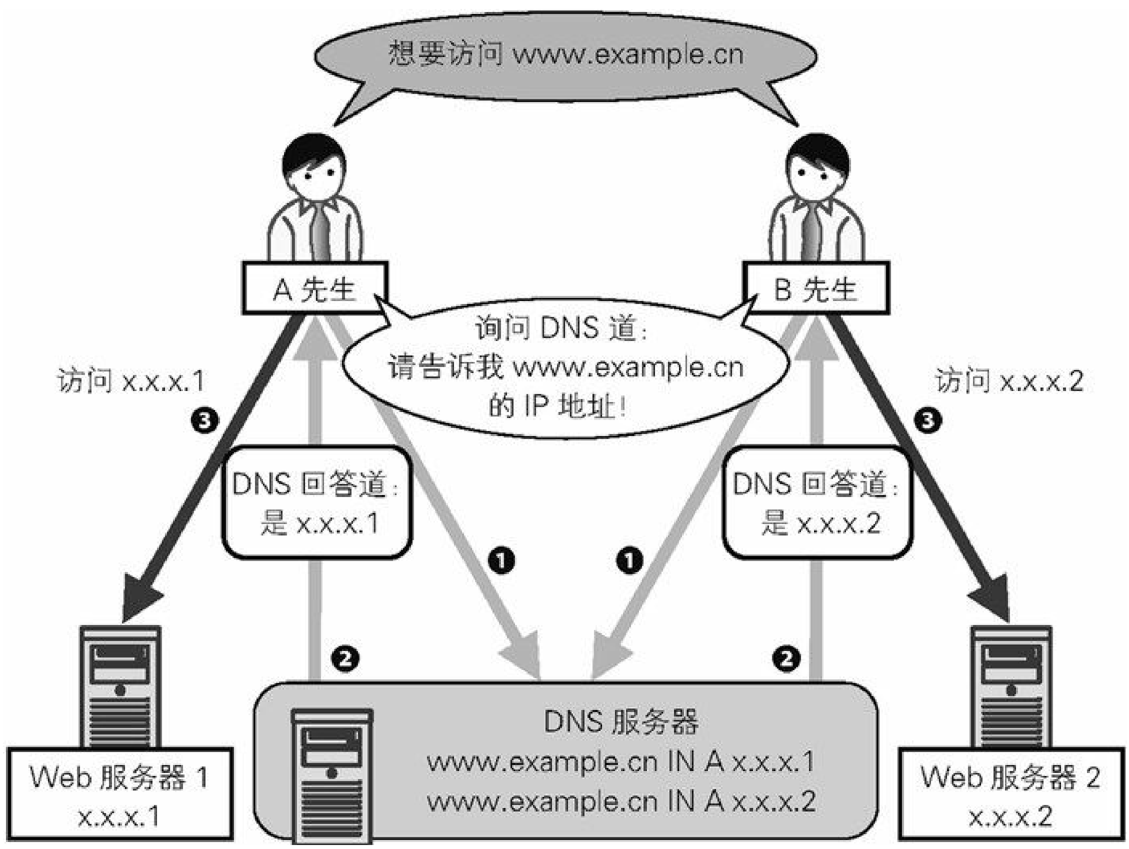


图 **1.2.1**　**DNS** 轮询

在 DNS 服务器中，若同一个域名上提交了多个记录（Record），则每

次访问时 DNS 服务器都会解析到不同的 IP 地址。如果能够利用好这一特性，就能把需要处理的内容分配到多台服务器上进行处理。这是相对比较简易的负载分发的实现方式，但还存在以下问题：

- 必须取得需要轮询的目标服务器的全局地址

  为了实现多台服务器的负载分发，取得这些服务器的 IP 地址是利用服务（线路）的前提

- 并不一定能实现均等的分发

  在手机站点中，这个问题被逐渐暴露了出来。通常来自手机的访问会经由称为行业网关的代理服务器，由于代理服务器会将域名解析的结果缓存一段时间，所以经由该代理服务器的访问请求就会被解析到同一台服务器上，因此就可能无法实现均等分配需要处理的请求，而只令某台服务器集中处理。另外在 PC 平台上的网页浏览器也会缓存这些 DNS 解析的结果，这就令均等分配更加不可能。虽说将 DNS 记录的 TTL（Time To Live，DNS 记录的缓存时间）修改得短一些多少可以改善这个问题，但这并不说只要修改 TTL，令 DNS 缓存频繁释放，就能解决根本问题

- 无从得知服务器宕机

  DNS 服务器不能知晓 Web 服务器的负载或连接数等信息，因此无法根据服务器的反馈调节分配。当 Web 服务器的负载逐步加重，响应时间也变得很慢时，DNS 也无从得知连接数是否已经超出服务器能够处理的范围。也就是说，即便服务器已经因为某种原因宕机，DNS 服务器也不会注意到这些问题，只会继续进行负载分发处理。万一用户被分配到已经宕机的服务器，那么该用户要面对的将是出错的页面。DNS 轮询始终只是机械地做着负载分发的操作，而这并不属于冗余的操作，因此很有必要安装其他软件来进行健康检查、故障转移等工作

1.2.2　DNS 轮询的冗余拓扑结构示例

在如图 1.2.2 所示的拓扑结构中，两台 Web 服务器都被分配了 VIP（虚拟 IP）。万一 Web1 停止工作，VIP1 就会映射到 Web2，其后所有的访问都将由 Web2 处理。相反，万一 Web2 停止工作，VIP2 就会映射到

Web1，自然其后所有的访问都将由 Web1 处理。正是因为 Web 服务器彼此之间互相协调，所以只要有能够正常工作的服务器，VIP 就会进行映射操作以维持服务的正常提供。

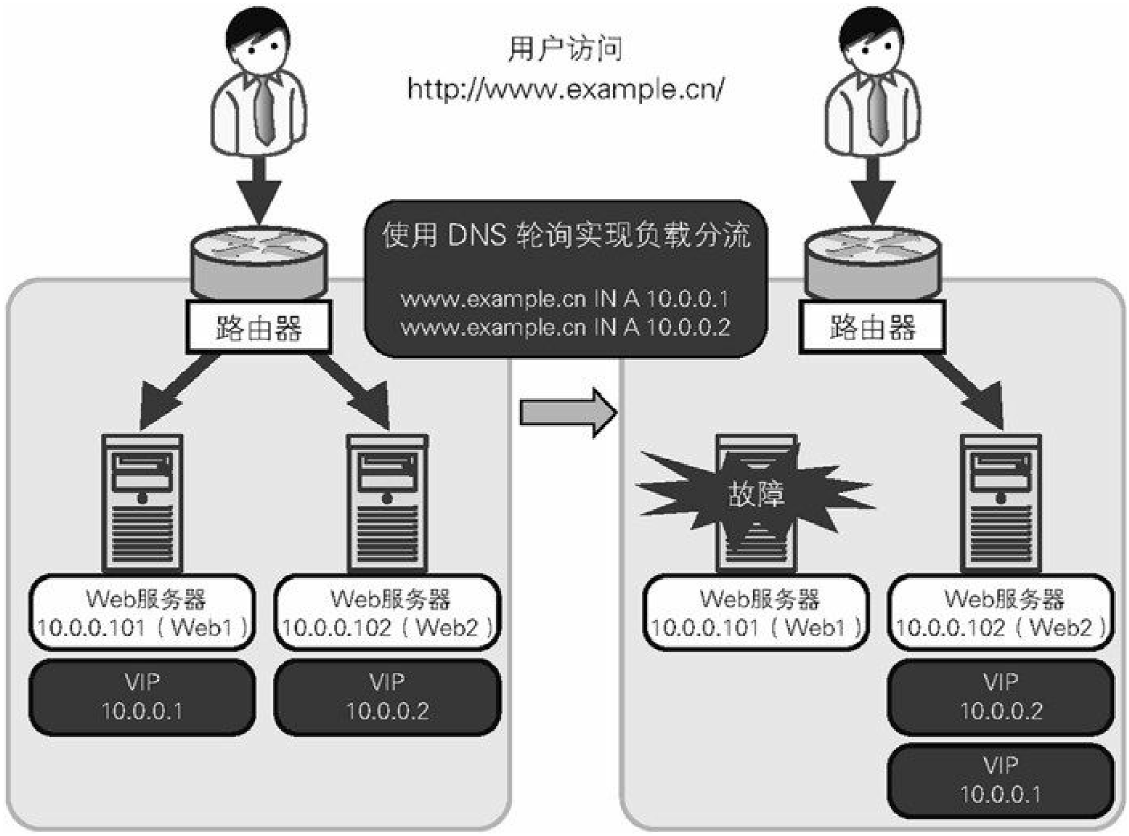


图 **1.2.2　DNS** 轮询的冗余拓扑结构示例

如果直接使用上文介绍的代码清单 1.1.1（failover.sh）来搭建这个拓扑结构，就可能会出现以下问题：

- 必须修改 **Web1** 与 **Web2** 的 **VIP** 设定值

  → 无法在两台服务器上使用同样的脚本

- 由于使用了 **ICMP** 监控，因此即便 **Web** 服务（**Apache** 等）停止，也无法触发故障转移

于是，我们将代码清单 1.1.1 的脚本修改成代码清单 1.2.1 的脚本。

代码清单 **1.2.1　failover2.sh**

```
#!/bin/sh
DEV="eth0"
VIP="10.0.0.1 10.0.0.2"

healthcheck() {
  for i in $VIP;do
    if [ -z "`ip addr show $DEV | grep $i`" ]; then
      if [ "200" -ne "`curl -s -I 'http://$i/' | head -n 1 | cut -f 2 -d '
        CIP="$i"
        return 1
      fi
    fi
  done
  return 0
}

ip_takeover() {
  MAC=`ip link show $DEV | egrep -o '([0-9a-f]{2}:){5}[0-9a-f]{2}' | head -
  ip addr add $CIP/24 dev $DEV
  send_arp $CIP $MAC 255.255.255.255 ffffffffffff
}

while healthcheck; do
  echo "health ok!"
  sleep 1
done
echo "fail over!"
ip_takeover
```

这样两台服务器就能使用同样内容的脚本了。这里并没有使用 ping 指令，而是通过使用 curl[5] 来进行健康检查，所以即使 Web 服务停止，也能启动故障转移，但是这样做仍会留下两个问题：

> [5] 命令行的 HTTP 客户端软件。 ***URL*** http://curl.haxx.se/

- 由于即使 **Web** 服务停止也无法释放 **VIP**，因此可能会造成 **IP** 地址冲突
- 一旦启动故障转移，该脚本就会停止运行

考虑到以上两点，还需要对脚本做些修改，如代码清单 1.2.2 所示。

代码清单 **1.2.2　failover3.sh**

```
#!/bin/sh
DEV="eth0"
VIP="10.0.0.1 10.0.0.2"

ip_add() {
  MAC=`ip link show $DEV | egrep -o '([0-9a-f]{2}:){5}[0-9a-f]{2}' | head -
  ip addr add $1/24 dev $DEV
  send_arp $i $MAC 255.255.255.255 ffffffffffff
}

ip_del() {
  ip addr del $1/24 dev $DEV
}

healthcheck() {
  for i in $VIP;do
    if [ "200" -ne "`curl -s -I 'http://$i/' | head -n 1 | cut -f 2 -d ' '"
      if [ -z "`ip addr show $DEV | grep $i`" ]; then
        ip_add $i
      else
        ip_del $i
      fi
    fi
  done
}

while true; do healthcheck;sleep 1;done
```

代码清单 1.2.2 的修改增加了以下内容：在 VIP 的健康检查失败的情况下，若为自行分配的地址，则可视为本机的 Web 服务发生了异常，其后释放 VIP ；相反，若不是自行分配的地址，则可视为对方的 Web 服务发生了异常，从而自行将 VIP 映射。另外，在本脚本中，无论如何映射 VIP，脚本都不会停止运行。

1.2.3　还想更轻松地扩充系统 **……**负载均衡器

根据以上介绍，无论哪方服务器的 Web 服务发生了异常，都能够正确地启动故障转移。而要想使用 DNS 轮询在实现负载分发的同时也实现

冗余，就需要花费很多心力。随着服务器数量的增加，系统会越来越复杂，设置 DNS 轮询的冗余搭建也会变得越来越难。如果在 3 台服务器上使用代码清单 1.2.2 的脚本，就会出现以下问题：

- 在某个服务器宕机后无法确定具体要交接到哪个服务器
- 根据启动故障转移的时机，有时可能会让两台服务器分配到同样的 **IP** 地址
- 一旦服务器停止运行，修复这台服务器的工作会变得很困难

虽说通过进一步完善脚本或者配合使用其他的软件应该也能解决这些问题，但我们还是希望可以更轻松地扩充系统。另外，在 Web 服务器的拓扑结构中还是尽量不要运行其他软件。在下面一节中，我们将介绍负载均衡器，通过引入负载均衡器，以上问题就会迎刃而解。

1.3　实现 **Web** 服务器的冗余 **——**基于 **IPVS** 的负载均衡器

1.3.1　**DNS** 轮询与负载均衡器的不同点

负载均衡器（Load Balancer，也称负载分发或负载分流设备）能够将向同一 IP 地址发出的请求分发到多台服务器进行处理。而 DNS 轮询则需要分别向 Web 服务器分配不同的 IP 地址（全局地址）。因此使用负载均衡器，能够节省全局地址的使用。使用 DNS 轮询的情况下，在架设 Web 服务的冗余结构时会费心费力，而负载均衡器则无需这样的操作。

- 负载均衡器的行为

  负载均衡器会被作为虚拟服务器使用，它拥有提供服务的全局地址。通过将来自客户端的请求中转到真正进行处理的 Web 服务器（下文统称为真实服务器），就像是 Web 服务器一样运行

- 负载均衡器的功能

  负载均衡器是从多台真实服务器中选出其中一台，并交由其进行处

理。负载均衡器不会选择健康检查失败的服务器，而一定会选择健康检查成功的服务器。因此无论哪台服务器停止运行，只要有一台服务器正常工作，都不会影响到整个服务的正常提供

- 引入负载均衡器的门槛

  人们可能会对负载均衡器抱有“负载均衡器 = 昂贵的设备”这样的印象，还有可能会担心自己能否使用好这套设备等，这些顾虑都会提高引入负载均衡的门槛

关于引入负载均衡器的门槛，确实专用的服务器产品会比较贵，每月的最低消费等费用也比较高。而且万一发生故障，还有必要联系研发商以获得该负载均衡器的技术支持。根据使用情况，还有可能需要升级防火墙等，因此也不能中断维护合同来缩减运营经费。在确保一定程度的盈利之前，难以下决心引入负载均衡器的情况很常见。但是也可以选择不使用专用服务器，而使用 OSS（OpenSource Software，开源软件）来搭建，并由自己来运营。接下来将正式讲解如何自行搭建生产环境中的负载均衡器。

1.3.2 IPVS ……基于 Linux 的负载均衡器

即使不安装特别的软件，Linux 也可以作为路由器（网络设备）使用。而且系统的防火墙也有很多实用的功能，例如分组过滤（Packet Filtering）技术等众多的网络功能。提供负载均衡功能的 **IPVS** （IP Virtual Server）模块也包含在其中。

负载均衡器的种类与 **IPVS** 的功能

接下来说明负载均衡器的种类。负载均衡器的种类按大的方向可分为四层交换机（L4 switch）与七层交换机（L7 switch）两种 [6] 。四层交换机进行的是传输层信息的处理，可以根据 IP 地址或端口号，指定作为分发目的地的服务器。而七层交换机进行的则是应用层信息的处理，可以根据从客户端请求的 URL，指定分发目的地的目标服务器。

> [6] L4 与 L7 在 OSI 参考模型中分别对应第四层（传输层，Transport Layer）与第七层（应用层，Application Layer）。

安装了 IPVS 就“相当于拥有了四层交换机的功能”，而七层交换机这里

暂时不会用到。

在本书的讲解中，若没有特别说明，均可认为是四层交换机的负载均衡器 [7] 。另外，一般情况下说到负载均衡器，大多默认指“四层交换机”。

[7] 在使用反向代理的情况下，也有可能实现一部分七层交换机的功能。关于反向代理的详情，请参考 2.1 节。

1.3.3　调度算法

将处理分发到真实服务器时，如果平均地分流到所有服务器，在不同配置的服务器混合使用的环境下就有可能造成服务器负载不平衡。在 IPVS 中有一种名为“调度算法”（Scheduling Algorithm）的调度器，必要的时候能够根据环境选择适当的算法。表 1.3.1 是主要的算法一览表。

表 **1.3.1**　主要的算法

名称
行为
rr（**round-robin**，轮询）调度
该算法不考虑别的因素，单纯以轮叫的方式依次请求真实服务器。因此处理会被均等地分发给所有的服务器
wrr（**weighted round-robin**，加权轮询）调度
与上述的rr有些类似，但该算法引用了一个加权值来控制分发的比率。加权值越大，服务器被选择的概率就越高，因此为了让处理性能较高的服务器承受更多请求，只需增大其加权值即可
lc（**least-connection**，最小连接）调度
该算法是将新的连接请求分配到当前连接数最少的服务器。在大部分的情况下使用这个算法都没有问题。在不知道到底要选择哪种算法合适的情况下，选择这个准没错
wlc（**weighted least-connection**，加权最小连接）调度
与上述的lc有些类似，不过该算法引入了加权值。由于该算法通过“（连接数+1）÷加权值”算出了最小因数的服务器，所以会更有效地分配连接。与wrr一样，加大高性能的服务器的加权值是比较好的选择
sed（**shortest expected delay**，最短预期延时）调度
该算法会选择响应速度最快的那台服务器。但它并没有监测传送数据分组到目标服务器的应答时间，而是选择状态是ESTABLISHED的连接数（下文统称为活动连接数）最少的服务器。其行为基本上和上述的wlc一样，但wlc会把除ESTABLISHED以

外的状态（TIME_WAIT或FIN_WAIT等）的连接数计算在最小因数中，这点即是wlc与sed不同的地方
nq（**never queue**，不排队）调度
与上述的sed算法类似，该算法会最优先选择活动连接数为0的服务器

针对真实服务器的不同配置，引入了“加权值”（Weight）这一参数，可以根据需要对加权值进行合理的设定。在某些算法中，该加权值越大就表示服务器的处理能力越高，据此可以调节分流比率。在表 1.3.1 中的算法行为一栏中，对每个算法如何选择真实服务器一一进行了说明。

IPVS 中还包含有表 1.3.1 中没有列出的调度算法。通过将 IPVS 和 透明代理（Transparent Proxy）或高速缓存服务器等并用，可以优化性能。虽然这里不会用到，不过我们还是将这些算法进行了整理，如表 1.3.2 所示。

表 **1.3.2**　其他的调度算法

名称
行为
sh（**source hashing**，源地址散列）调度
对发出请求的IP地址（即源地址）计算散列值（Hash Value），并通过该值选择具体分发到哪个真实服务器
dh（**destination hashing**，目标地址散列）调度
对需要接收请求的目标IP地址计算散列值，并通过该值选择具体分发到哪个真实服务器
lblc（**locality-based least-connection**，基于局部性的最小连接） 调度
在连接数没有超过加权值指定的值时，将选择同一台服务器。若是超过了加权值指定的值，则选择其他的服务器。当所有服务器的连接数都超过加权值指定的值时，将选择最终所选的那台服务器
lblcr（**locality-based least-connection with replication**，带复制的基于局部性最小连接） 调度
与上述的lblc有些类似，当所有服务器的连接数都超过加权值指定的值时，将选择连接数最少的那台服务器

1.3.4　使用 IPVS

可以通过以下软件使用 IPVS 的功能：

- ipvsadm ***URL*** http://www.linuxvirtualserver.org/software/ipvs.html
- keepalived ***URL*** http://www.keepalived.org/

ipvsadm

ipvsadm 是由 IPVS 的开发商提供的命令行工具[8]。除了定义虚拟服务器及分配真实服务器等功能外，也可以确认设定内容和连接状态，另外也能将传输率等统计信息显示出来。

> [8] 其地位就相当于 netfilter 组件与 iptables 命令之间的关系。

keepalived

keepalived 是用 C 语言编写的守护程序。可以根据配置文件（/etc/keepalived/keepalived.conf）的内容来搭建 IPVS 的虚拟主机。其功能主要表现在：对真实服务器进行健康检查，并自动将已经宕机的服务器排除在负载均衡所分配的范围外；如果所有的真实服务器全数宕机，则会显示“服务器繁忙”等消息，即具备 sorry_server 功能。

截至 2014 年 11 月，最新的版本是 keepalived-1.2.13，该版本支持以下健康检查。

- **HTTP_GET**：通过发送 **HTTP** 的 **GET** 请求来确认目标服务器的应答情况
- **SSL_GET**：通过发送 **HTTPS** 的 **GET** 请求来确认目标服务器的应答情况
- **TCP_CHECK**：通过测试 **TCP** 连接是否正常来确认目标服务器的情况
- **SMTP_CHECK**：通过发送 **SMTP** 的 **HELO** 命令来确认目标服务器的应答情况
- **MISC_CHECK**：通过执行外部命令所返回的结束代码来确认目标服务器的健康状况

1.3.5　搭建负载均衡器

接下来使用 keepalived 搭建如图 1.3.1 所示的系统。这里将 10.0.0.1 作为服务器使用的全局 IP 地址。在该负载分发的拓扑结构中，当客户端访问http://10.0.0.1/时，该请求就会被转到 Web1 与 Web2 进行处理。其他详细的配置参数如下。

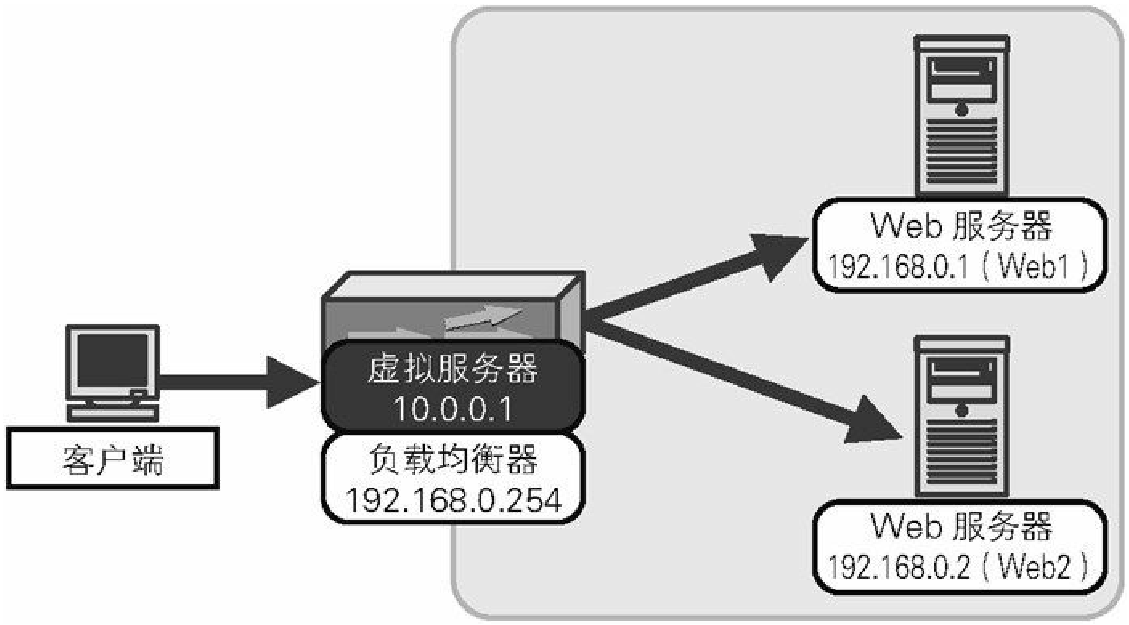


图 **1.3.1**　使用负载均衡器进行负载分流

- 调度算法：**rr**（**round-robin**）
- 健康检查的类别：**HTTP_GET**
- 健康检查的页面：**http://health/health.html**
- 健康检查成功的条件：状态代码返回 **200**
- 健康检查的超时时间：**5** 秒

代码清单 1.3.1 是将以上参数作为 keepalived 的配置文件进行编写的。

代码清单 **1.3.1**　**keepalived** 的配置

```
virtual_server_group example {
```

```
  10.0.0.1 80
}
virtual_server group example {
  lvs_sched   rr
  lvs_method  NAT
  protocol    TCP
  virtualhost health
  real_server 192.168.0.1 80 {
    weight 1
    HTTP_GET {
      url {
        path /health.html
        status_code 200
      }
      connect_port    80
      connect_timeout 5
    }
  }
  real_server 192.168.0.2 80 {
    weight 1
    HTTP_GET {
      url {
        path /health.html
        status_code 200
      }
      connect_port    80
      connect_timeout 5
    }
  }
}
```

配置 **Web** 服务器

在启动 keepalived 前，需要确认 Web 服务器的配置，必要的操作有下面 3 点：

- 设置默认网关为 **192.168.0.254**
- 设置健康检查页面
- 设置用于确认运行情况的页面

在此拓扑中，来自客户端的请求与来自真实服务器的响应都必须经由负载均衡器，因此需要事先设置各 Web 服务器的默认网关为负载均衡器的 IP 地址。

Keepalived 会对真实服务器进行健康检查。这里访问 `http://health/health.html`，并检查是否会返回 200 的状态代码，因此有必要在每个 Web 服务器上都设置一个用来做健康检查的页面。

另外还需要准备好确认运行情况的页面。在确认运行情况时，为了更容易地把握具体分流到了哪个服务器，应该特意将 index.html 写入为不同的内容以示区别，这里将 index.html 写入主机名（Web1、Web2）。

启动 **keepalived**

请把代码清单 1.3.1 放到 /ect/keepalived/keepalived.conf 下，然后启动 keepalived，IPVS 的虚拟服务器就搭建好了。如图 1.3.2 所示，可以使用`ipvsadm` 命令确认虚拟服务器。在图 1.3.2 中，代码清单将连接到 10.0.0.1:80 的请求，分流到 192.168.0.1:80 与 192.168.0.2:80 两台主机上进行处理。

```
# ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=1048576)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.0.0.1:80 rr
  -> 192.168.0.1:80               Masq    1      0          0
  -> 192.168.0.2:80               Masq    1      0          0
```

图 **1.3.2**　确认虚拟服务器

确认负载分流

从客户端访问 http://10.0.0.1/，会得出如图 1.3.3 所示的结果。每次访问都会交替连接到 Web1 与 Web2，从而就可以确认负载分流的具体执行情况。

```
$ curl 'http://10.0.0.1/'
```

```
Web1

$ curl 'http://10.0.0.1/'
Web2

$ curl 'http://10.0.0.1/'
Web1

$ curl 'http://10.0.0.1/'
Web2
```

图 **1.3.3**　确认负载分流

确认冗余的拓扑结构

下面在 Web2 宕机的情况下再次尝试访问。从图 1.3.4 中能够看出，可以正常连接到 Web1 的主机。所以能够确认：在冗余的拓扑结构下，即使 Web2 停止工作也不会产生错误，依旧可以正常访问。

```
$ curl 'http://10.0.0.1/'
Web1

$ curl 'http://10.0.0.1/'
Web1

$ curl 'http://10.0.0.1/'
Web1

$ curl 'http://10.0.0.1/'
Web1
```

图 **1.3.4**　确认冗余的拓扑结构

1.3.6　四层交换机与七层交换机

前文提到了负载均衡器分为四层交换机与七层交换机两种。虽说都是进行负载分发，但两者的处理内容却有很大的差异。四层交换机通过解析

TCP 头等协议头的内容，来决定分流的目的地；而七层交换机则通过解析软件应用层的内容，来决定分流的目的地。

图 1.3.5 表现了四层交换机与七层交换机的不同点。在四层交换机中，客户端（Web 浏览器）的通信目标是真实服务器。而在七层交换机中，负载均衡器和客户端会先通过 TCP 会话进行握手。也就是说，每次访问都会经过：客户端<=>七层交换机中黑色的框<=>七层交换机中白色的框<=>真实服务器（图中为两台）。

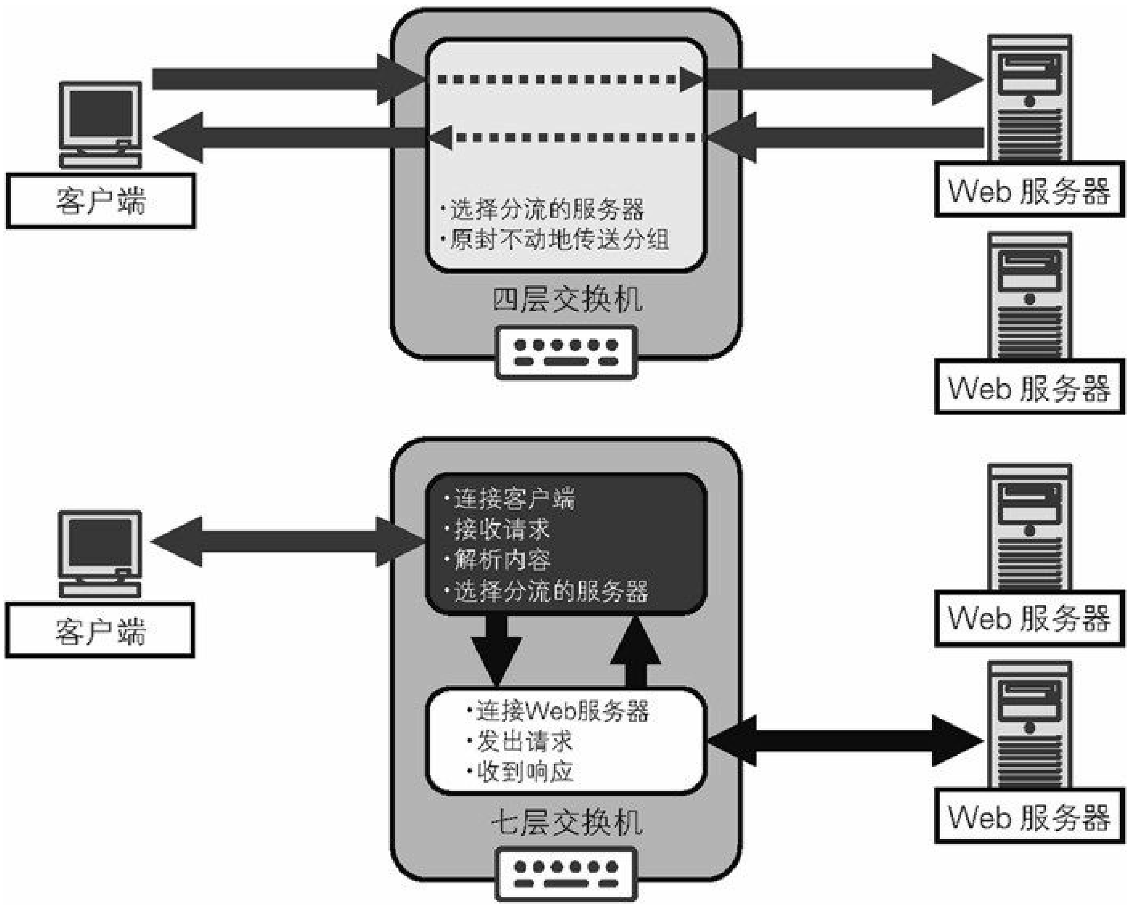


图 **1.3.5**　四层交换机与七层交换机的不同点

四层交换机与七层交换机的特点可以简明地总结如下：

- 追求设定的灵活性就用七层交换机
- 追求性能就用四层交换机

专栏

七层交换机的灵活设定

七层交换机可以将类似`http://example.cn/*.png`这样的图片文件发送的请求分配到图片专用的服务器进行处理。而对于像`http://example.cn/hoge?SESSIONID=xxxxxxxx`这样包含会话ID的请求，七层交换机还可以将同一会话ID的请求分配到同一台服务器进行处理。

也就是说，类似于请求目标URL等应用软件协议的内容，可以被作为选择真实服务器时的条件使用。相反，负载均衡器不能识别的协议，则自然无法实现负载分流。比如在对SMTP进行负载分流时，虽说理论上可以通过七层交换机实现“将特定收件人的邮件发送到特定的服务器”，但若是负载均衡器不支持SMTP，那就无法使用。

如何决定将什么样的协议根据什么样的规则来进行分流呢？虽说可以依靠负载均衡器所提供的功能，但“使用七层交换机就万事大吉了”这样的想法是要不得的。在选定七层交换机时，将想要做的工作都准确无误地确认好，这是很重要的。

1.3.7　四层交换机的 NAT 模型与 DSR 模型

下面对四层交换机的性能优势做个介绍。请先思考一下四层交换机的哪些模型造就了该拓扑结构的性能优势。

四层交换机可以利用图 1.3.1 中搭建的 NAT（Network Address Translation，网络地址转换）模型，为了追求更高的性能，也可以搭建 DSR（Direct Server Return，直接服务器返回）模型。DSR 与 NAT 的不同点请参考图 1.3.6。

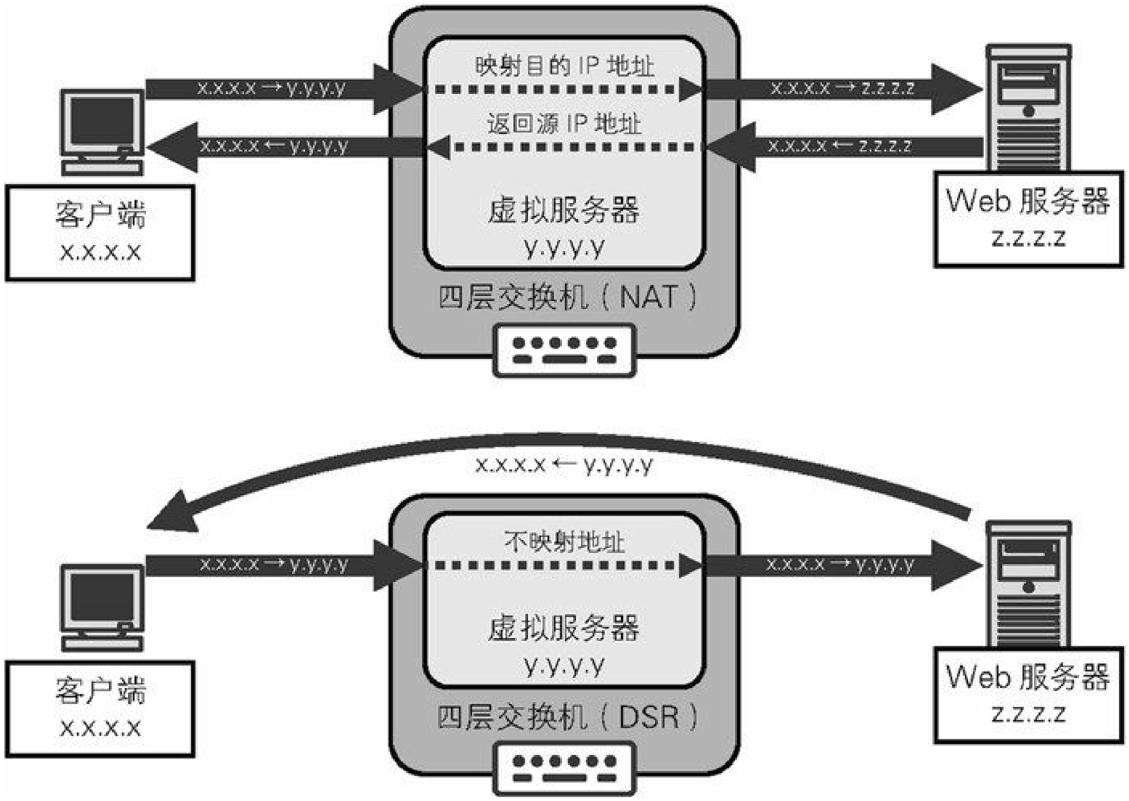


图 **1.3.6**　**DSR**（下图）与 **NAT**（上图）的不同点

在 NAT 模型下，四层交换机从客户端收到数据分组后就修改分组的目的 IP 地址，并将分组转发到真实服务器上。因此在真实服务器接收到应答数据分组后，需要特别返回源 IP 地址。而在 DSR 模型下，IP 地址并不会被映射。四层交换机从客户端接收到分组后，不做任何修改就直接将分组从路由器发送到真实服务器上。在这种情况下，由于没有必要对应答的分组返回 IP 地址，因此真实服务器即使不经由四层交换机也能够返回应答。

若担心负载均衡器出现瓶颈问题，或者需要能够承受高流量的负载分发环境，这里推荐使用 DSR 模型。顺带一提，在使用 keepalived 来搭建 DSR 时，请将 lvs_method 设定为“DR”（请注意在这里不应该设定为“DSR”）。

但在 DSR 模型中，发送给虚拟服务器的分组（全局 IP 分组）没有做任何修改就原样到达了真实服务器，由于真实服务器不能处理该全局 IP 分组，因此无法处理该请求。也就可以说，对于 NAT 模型的系统，只修改负载均衡器的设定是没有办法实现负载均衡的。

最简便的设定方式是，将虚拟服务器的 IP 地址映射到真实服务器的环回接口（Loopback Interface）上，另外还有一种方法是，使用 netfilter 的相关功能，将发给虚拟服务器的分组的本地源地址映射为私有的本地全球地址（即在本地范围内扩大地址的使用范围，以扩展地址的容量），这种方式称为 DNAT（Destination Network Address Translation，目的网络地址转换）。

1.3.8　同一子网下的服务器进行负载分流时需要注意的地方

至此，我们已经探讨了公网 Web 服务器的负载分流。其实负载均衡器的用途并不仅限于此，例如在信息发布系统中，从 Web 服务器到邮件服务器可能需要发送大量邮件。若只有一台邮件服务器来处理，想必会花不少时间，因此可以考虑使用负载均衡器将邮件服务器进行负载分流。类似这样的情况可以考虑参考图 1.3.7，但该拓扑结构有几点需要注意的地方。

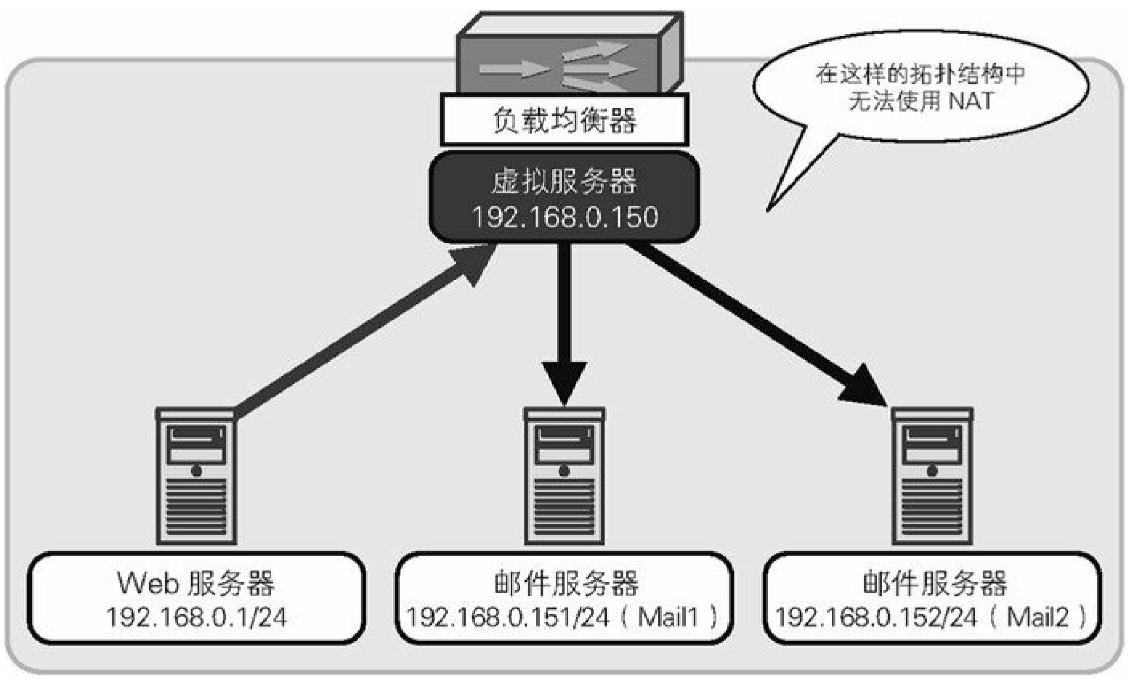


图 **1.3.7**　同一子网下的负载分流

在同一子网需要负载分流时，不能使用 NAT 模型。NAT 模型下会映射负载均衡器的源 IP 地址，比如在邮件服务器中，接收到的分组是从源地址 192.168.0.1 发送到目的地址 192.168.0.151 的，因此该邮件服务器返回的应答分组的源地址是 192.168.0.151，目的地址是 192.168.0.1。正

因为源地址 192.168.0.151 与目的地址 192.168.0.1 是同一子网下的 IP 地址，因此不会将分组发往负载均衡器，而直接从源地址发送到 Web 服务器。因此该 NAT 模型中源地址的 IP 地址尚未被正确转发到目标服务器，进而就造成了不能正常完成通信。

该情况下，使用上文介绍的 DSR 模型是比较好的选择。因为 DSR 模型不用映射负载均衡器的 IP 地址，所以就算邮件服务器直接向 Web 服务器返回应答也完全没有问题。

专栏

基于**Linux**平台的七层交换机

基于Linux平台搭建七层交换机所使用的软件还在不断开发中，以下介绍两款软件：

- UltraMonkey-L7

  URL http://sourceforge.jp/projects/ultramonkey-l7/

- Linux Layer7 Switching

  URL http://zh.sourceforge.jp/projects/freshmeat_linux-l7sw/

UltraMonkey-L7于2008年1月发布了1.0.1-0版本。截至目前（2014年11月），已经更新到了3.1.1-1版。现在使用UltraMonkey-L7已经可以搭建稳健的七层交换机，该软件的实用性也非常让人满意。

而Linux Layer7 Switching虽然于2007年1月发布了0.1.2版本,但似乎之后该软件并没有更新，该软件有多大的实用性现在还是未知数。

1.4　路由器及负载均衡器的冗余

1.4.1　负载均衡器的冗余

到目前为止，我们都只在介绍 Web 服务器的冗余，还尚未提及负载均衡器的冗余。如果仅有的一台负载均衡器发生故障的话，那么所有的服务都会停止。虽说可以使用冷备份，为预防故障事先多准备一台负载均

衡器作为备用，但是当故障发生时若没有人力干预，依旧无法及时恢复运行。

本节将介绍路由器及负载均衡器的故障转移的方法。

1.4.2　虚拟路由器冗余协议（VRRP）

现在市面上有很多包含冗余功能的路由器与负载均衡器的产品。以前由于产品之间安装的冗余实现各不相同，因此基本上都使用厂商独有的协议。

当然不同的协议间不能相互兼容，这会造成很多不便，Cisco 公司以 **HSRP** （Hot Standby Routing Protocol，热备份路由器协议）为基础，制定出了不依赖于厂商的冗余协议，即 **VRRP** （Virtual Router Redundancy Protocol，虚拟路由器冗余协议）。很多路由器及负载均衡器都采用了在 RFC 3768[9] 中定义的 VRRP。上一节中介绍的 keepalived 也使用了 VRRP，这样只需再搭建一台负载均衡器，并在 keepalived 添加相关的设定，就可以实现冗余。

> [9] ***URL*** http://www.ietf.org/rfc/rfc3768.txt

1.4.3　VRRP 的拓扑模型

路由器与负载均衡器的故障转移的原理，与 1.1 节中讲解 Web 服务器的故障转移流程时提到的“健康检查”“IP 地址的映射”几乎是一样的，即检查节点是否正常工作，万一出问题的话，备用节点就会映射 VIP（虚拟地址），发生故障转移。

在介绍 VRRP 的搭建及设定前，我们首先对 VRRP 常用的规则、术语，以及其行为做了如下整理。使用 VRRP 时，请留心这些关键字：“VRRP 报文”“虚拟规则 ID”“优先顺序”“抢占模式”“虚拟 MAC 地址”。

VRRP 报文

所谓健康检查，一般是指定期对监控对象的设备发出一些请求，确认这些请求是否取得了应答。而 VRRP 则通过相反的途径监控主节点的运行，即 VRRP 的主节点会定期将 **VRRP** 报文 （VRRP Packet）不断地发

送到多点传送地址（224.0.0.18，Multicast Address）。由于 VRRP 报文类似于“播报”主节点可用的信息，因此也被称为“组播信息”（Advertisement)。图 1.4.1 是 VRRP 报文的格式，该报文对以下三项数据进行了编排：

- **IP Address**（虚拟 **IP** 地址，即 **VIP**）
- **Virtual Rtr ID**（虚拟路由器 **ID**）
- **Priority**（优先顺序）

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version| Type  | Virtual Rtr ID|   Priority    | Count IP Addrs|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Auth Type   |   Adver Int   |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         IP Address (1)                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            .                                  |
|                            .                                  |
|                            .                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         IP Address (n)                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Authentication Data (1)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Authentication Data (2)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

图 **1.4.1** **VRRP** 报文的格式 ※

※ 引自 http://www.ietf.org/rfc/rfc3768.txt。

备用节点通常会在正确接收到 VRRP 报文时进入待机状态，若一段时间没有收到 VRRP 报文的信息，备用节点就会认为主节点已经宕机，从而启动故障转移。因此在 VRRP 中，备用节点不会主动地确认主节点的状态。

虚拟路由器 **ID**

向事先决定的组播地址（224.0.0.18）发送 VRRP 报文时，该组播地址从头到尾都不会改变。如图 1.4.2 所示，在同一网络上同时设置有多台负载均衡器的情况下，所有的 VRRP 报文都会发向同一地址。这看起来似乎会引起错误操作，但实际上 VRRP 中使用了名为虚拟路由器 **ID** 的参数，可以分辨实例，因此不会出现问题。如图 1.4.2 所示，在负载均衡器 A、B 与负载均衡器 C、D 中，只需对虚拟路由器 ID 做出合理设定，就能确保图 1.4.2 的拓扑模型可以正常工作。

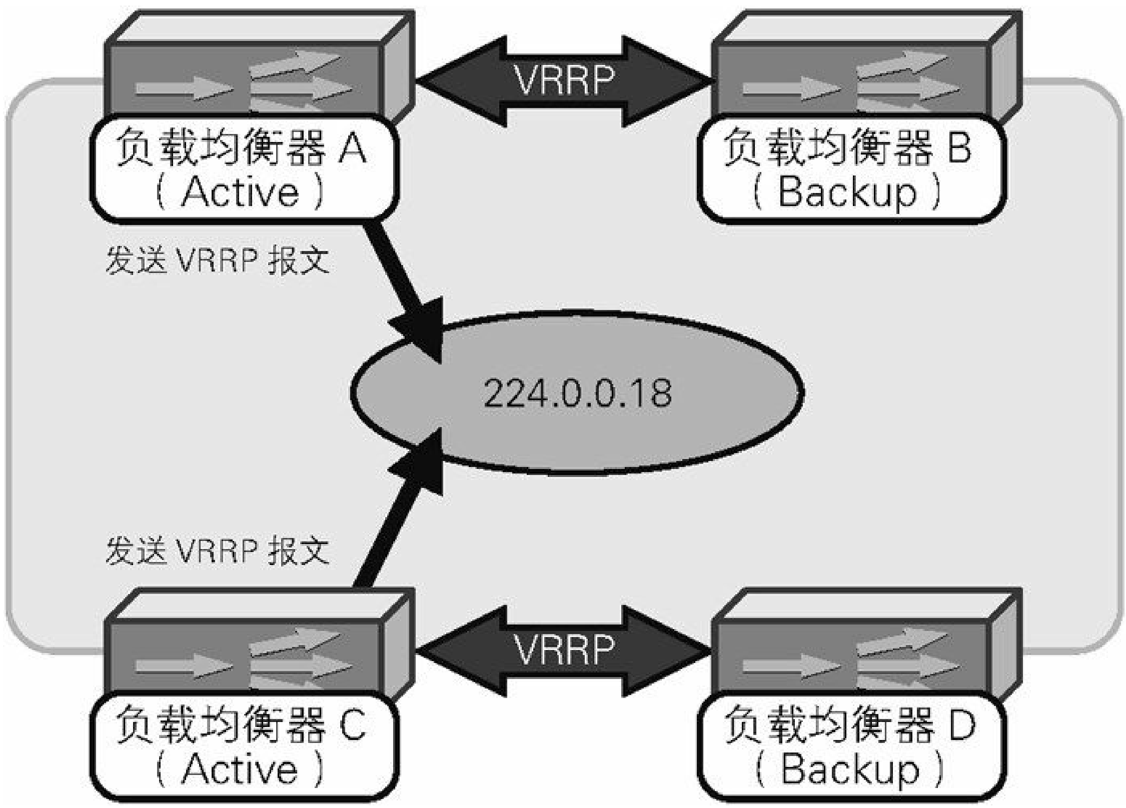


图 **1.4.2**　在同一网络上设置多台负载均衡器

优先顺序

在 VRRP 的拓扑结构示例中，经常可以看到拓扑结构由 Active/Backup 两台设备构成，但在实际使用中也有可能同时拥有 100 台以上的备用节点，这样就可能会造成一个问题：当两台以上的备用节点同时工作时，若是主节点出现问题，那么主节点具体要漂移到哪个备用节点呢？

在 VRRP 中，会为各节点设定优先顺序（Priority）的值，各节点在接收不到 VRRP 报文时，会自行发出 VRRP 报文。由于 VRRP 报文中有事先设定的优先顺序值，因此通过比较该值，就可以迅速了解到比本节点优先值高的其他节点是否存在。万一有其他节点的优先值比本节点高，则其他节点会比本节点优先升格为主节点。虽然是非常简单的操作，但通过设定节点的优先顺序，在漂移到备用节点时，就可以从优先顺序高的节点依次进行，使优先顺序高的备用节点作为主节点使用。

抢占模式

在 VRRP 的默认设定中，当比现有的主节点优先顺序高的节点启动时，会发生故障转移。也就是说可以认为优先顺序高的节点一定是主节点。这个行为可以通过设定抢占模式（Preemptive Mode）进行改变。将抢占模式设为无效的情况下，只要主节点正常工作，即使其他节点的优先级比主节点高，也不会出现故障转移。

根据情况的不同，选择的抢占模式也不一样。比如主节点的情况已经不乐观，设备很可能会频繁重启，这时将抢占模式设为无效是比较好的选择。相反，如果是为了防止操作失误而希望在两节点都可以选择时，一定要漂移到特定的节点，则将抢占模式设为有效会是比较好的选择。

虚拟 **MAC** 地址

VRRP 中除了定义了虚拟 IP 地址，还定义了虚拟 **MAC** 地址 。为了完成映射，在故障保护时，不仅需要映射 IP 地址，还需要映射 MAC 地址。如果在不映射 MAC 地址的情况下映射 IP 地址，通信目标下所有设备的 ARP 缓存表都需要更新。为此，一般使用 1.1 节介绍的 gratuitous ARP 进行 ARP 请求，但在以太网（Ethernet）的操作环境中，无法保证所有的设备都能正常收到 ARP 请求。万一哪个设备的 ARP 缓存表没有更新，则直到该设备的 ARP 缓存更新为止，都没办法与这台设备进行通信。

为此，在 VRRP 中映射 MAC 地址时，需要对通信目标的 ARP 请求的更新进行特别处理。例如在 RFC 3768 中，在进入主节点状态前会发出 gratuitous ARP，其实这样做的目的不是为了更新通信目标的 ARP 缓存表，而是为了更新二层交换机中 MAC 地址的学习状态。

1.4.4 安装 **keepalived** 时可能遇到的问题

在 keepalived 的 VRRP 中，由于不使用虚拟 MAC 地址，因此无法按照 RFC 3768 的方式进行安装。虽说在 Linux 平台下可以改变 MAC 地址，但因为无法拥有多个 MAC 地址，安装的 keepalived 中就无法使用虚拟 MAC 地址，所以在故障转移时就很可能存在 ARP 请求还尚未更新的设备。在这种情况下，直到 ARP 缓存清除为止，该设备都无法进行通信。

延迟发送 **gratuitous ARP**（**GARP**）

为了解决这个问题，keepalived 中有一个“grap_master_delay”的设定项目。keepalived 在主节点状态漂移后，紧接着就会发出 gratuitous ARP，但在这个瞬间大都存在网络不稳定的状况，例如可能流量太过于集中造成无法通信。keepalived 为了确保让通信目标能准确地接收到更新 ARP 的请求，会安排在等待数秒后再发送 gratuitous ARP。这个等待的时间可以在 grap_master_delay 中设定，默认值是 5 秒。

例如假设有这样一个系统：采用 STP（Spanning Tree Protocol，生成 树协议）[10] 搭建网络，在二层交换机宕机后，负载均衡器会启动故障转移。在二层交换机宕机的瞬间，STP 的收敛（Convergence）开始进行，这可能会造成数秒到数十秒的通信中断。由于在此期间发送的 gratuitous ARP 并没有实际传到其他节点，因此可以通过设定 grap_master_delay 在收敛完成后再发送 gratuitous ARP。在 keepalived 的 VRRP 实现中，因为存在与 RFC 3768 中所定义的内容不一样的地方，因此在实际网络环境中使用时，有必要验证是否能够正确启动故障转移。

> [10] IEEE 802.1D 规定了这种链路管理协议。

1.4.5　keepalived 的冗余

接下来，使用 keepalived 搭建出如图 1.4.3 所示的系统。图 1.4.3 中的 lv1 与 lv2 是在 Linux 中安装了 keepalived 的负载均衡器。代码清单 1.4.1 是 lv1 与 lv2 的 keepalived.conf 的设定 [11] 。表 1.4.1 中解释了各个参数的意义。

> [11] 代码清单 1.4.1 中省略了 IPVS（负载分流）的相关设定。

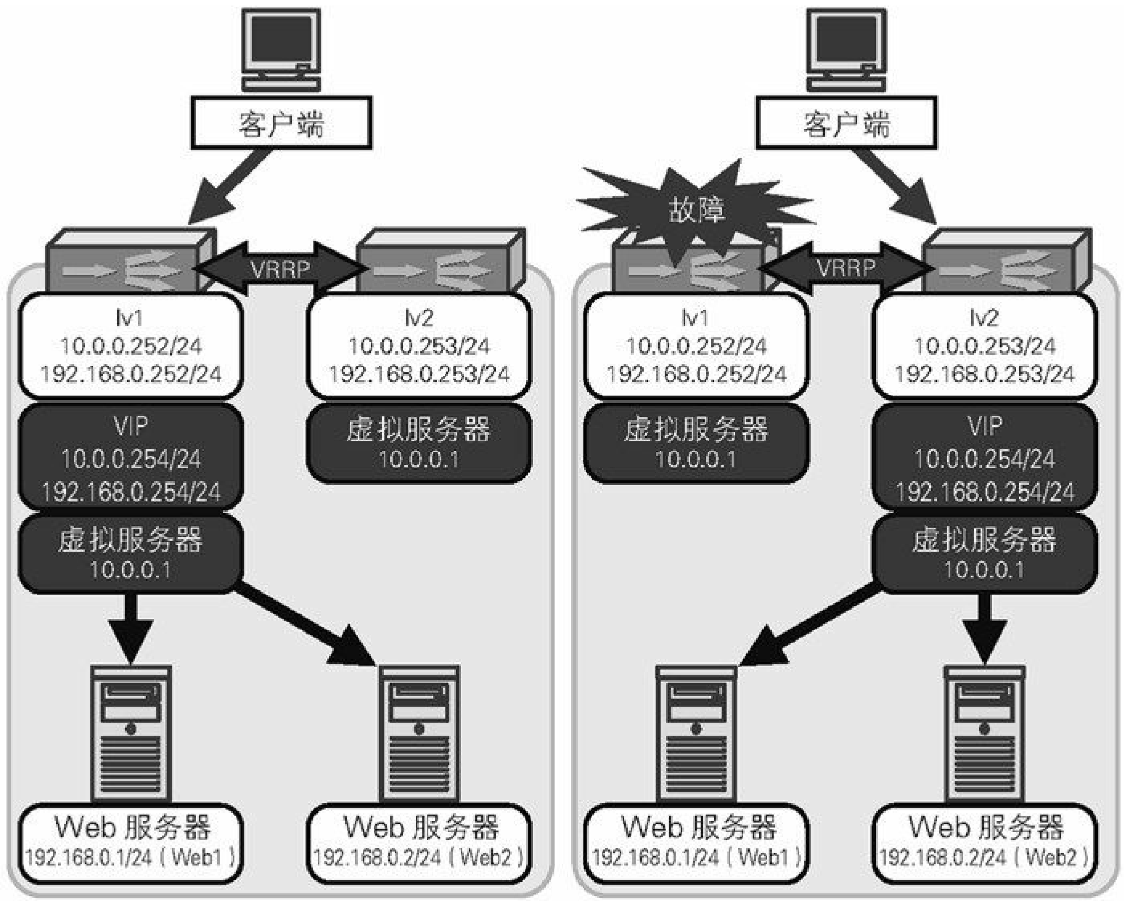


图 **1.4.3**　负载均衡器的冗余

代码清单 **1.4.1**　**VRRP** 的设定 ❶（**lv1** 的示例）

```
vrrp_instance VI {
  state MASTER
  interface eth0
  garp_master_delay 5
  virtual_router_id 200
  priority 101 ←在lv2中修改为100
  advert_int 1
  authentication {
      auth_type PASS
      auth_pass HIMITSU
  }
  virtual_ipaddress {
      10.0.0.254/24    dev eth0
      192.168.0.254/24 dev eth1
  }
}
```

表 **1.4.1**　各参数的意义

参数	说明
state MASTER	启动 keepalived 的时候，指定是作为 MASTER（主节点）启动还是作为 BACKUP（备用节点）启动
interface eth0	指定发出或接收 VRRP 报文的接口
garp_master_delay 5	指定从主节点状态漂移后，到再次发送 gratuitous ARP 的间隔秒数
virtual_router_id 100	虚拟路由器 ID。每个 VRRP 实例都要指定一个独一无二的值，可指定的范围是 0 到 255
priority 101	VRRP 优先顺序的设定值。在选择主节点的时候，该值大的备用节点会优先漂移为主节点
advert_int 1	VRRP 报文的发送间隔。以秒为单位来指定。默认值为 1 秒
virtual_ipaddress	VIP（虚拟地址）。书写格式如下，可以同时指定两个以上的虚拟地址 `<IPADDR>/<MASK>dev<STRING>`

确认 **VIP**

在 lv1 与 lv2 中启动了 keepalived 后，lv1 就被分配了 VIP（10.0.0.254 与 192.168.0. 254），但这无法通过`ifconfig` 命令来确认，可以通过如图 1.4.4 所示的`ip` 命令进行确认。

```
lv1:~# ip addr show eth0
```

```
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
  link/ether xx:xx:xx:xx:xx:xx brd ff:ff:ff:ff:ff:ff
  inet 10.0.0.252/24 brd 10.0.0.255 scope global eth0
  inet 10.0.0.254/24 scope global eth0

lv1:~# ip addr show eth1
3: eth1: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 1000
  link/ether xx:xx:xx:xx:xx:xx brd ff:ff:ff:ff:ff:ff
  inet 192.168.0.252/24 brd 192.168.0.255 scope global eth1
  inet 192.168.0.254/24 scope global eth1
```

图 **1.4.4**　确认 **VIP**

确认 **VRRP** 的运行情况

故障转移的运行情况的确认结果可总结如下：

❶ 关闭 **lv1 → lv2=Master ○**

❷ 启动 **lv1 → lv1=Master，lv2=Backup ○**

❸ 将 **lv1** 的 **eth0** 网线拔掉 **→ lv1=Backup，lv2=Master ○**

❹ 将 **lv1** 的 **eth0** 网线插回 **→ lv1=Master，lv2=Backup ○**

❺ 将 **lv1** 的 **eth1** 网线拔掉 **→ lv1=Master，lv2=Backup ×**

这样看来，在上述第 ❺ 条拔掉 eth1 网线时，情况似乎有些奇怪。原本在 lv1 的 eth1 链路失效时，应该是 lv1=Backup，lv2=Master。

分离 **VRRP** 实例

在以上的设定中，因为 VRRP 报文只有 eth0 传输数据，所以在 eth1 网线拔掉时自然无法检测出异常。如果想检测出多个实例的故障，就要如代码清单 1.4.2 所示，对每个接口的 VRRP 实例都做出定义。这里请注意“virtual_router_id”这个参数，在照搬 vrrp_instance 代码块中的代码时，请注意不要忘记替换相关的参数。因为 VRRP 实例是根据 virtual_router_id 区分的，所以请为每个实例指定一个独一无二的值。

代码清单 **1.4.2　VRRP** 的设定❷（**lv1** 的示例）

```
vrrp_sync_group VG {
  group {
    VE
    VI
  }
}
vrrp_instance VE {
  state MASTER
  interface eth0
  garp_master_delay 5
  virtual_router_id 200 ←为每个VRRP实例指定独一无二的值
  priority 101 ←lv2中修改为100
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass HIMITSU
  }
  virtual_ipaddress {
    10.0.0.254/24    dev eth0
  }
}
vrrp_instance VI {
  state MASTER
  interface eth1
  garp_master_delay 5
  virtual_router_id 201 ←每个VRRP实例独一无二的值
  priority 101 ←lv2中请将这里修改为100
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass HIMITSU
  }
  virtual_ipaddress {
    192.168.0.254/24 dev eth1
  }
}
```

同步 **VRRP** 实例

代码块 vrrp_sync_group 是为了实现多个 VRRP 实例的状态的同步所需

要进行的设定。例如，在对外实例（VE）作为备用节点的情况下，它所联动的对内实例（VI）也需要是备用的。这样在 lv1 中无论 eth0 或 eth1 哪边的网线出现异常，也都能够准确地执行故障转移。

1.4.6　**keepalived** 的应用

之前我们对使用 keepalived 的 VRRP 功能来实现故障转移的详情进行了说明。

根据需求的不同，keepalived 还有其他使用方法。比如在 1.1 节的图 1.1.6 的拓扑结构中，如果使用 keepalived 的话，搭建起来应该就能更加方便安全。keepalived 中还附带一个`--vrrp` 选项，据此可以独立地使用 VRRP 功能。读者可以先从搭建简单的冗余 SMTP 服务器开始，逐步加深对冗余的理解。

第 2 章　优化服务器及基础设施的拓扑结构 ——冗余、负载分流、高性能的实现

2.1　引入反向代理 ——Apache 模块

2.1.1　反向代理入门

第 1 章通过引入负载均衡器，实现了 Web 服务器的负载分流。因为 IPVS（LVS）等负载均衡器只在第四层传送分组，因此在该拓扑结构中，Web 服务器依然直接应答从客户端应用软件发来的请求。

通过在负载均衡器与 Web 服务器之间加入被称为反向代理 （Reverse Proxy）的服务器，可以实现更具弹性的负载分流。反向代理可通过 Apache 的 mod_proxy、mod_proxy_balancer 模块进行搭建。除 Apache 外，也可使用 lighttpd 或 Squid（稍后讲解）来进行搭建。

反向代理从客户端取得请求（必要的话并加以处理），将该请求转发到适当的 Web 服务器。Web 服务器收到请求后会像平时一样进行处理，并将结果返回反向代理。反向代理获取结果后，再将该处理结果的应答返回到客户端。

如图 2.1.1，反向代理的工作即为在客户端与 Web 服务器之间建立起类似代理的机制。一般的代理是代理处理 LAN → WAN 的请求，而反向代理则是代理处理 WAN → LAN 的请求，因此才被称为“反向”（Reverse）。

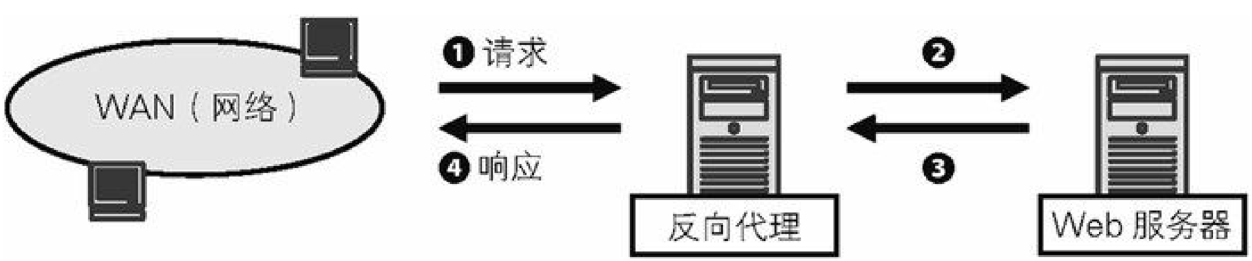


图 **2.1.1**　反向代理

使用反向代理，通过客户端请求，可以将网页的不同元素分配到不同的服务器专项处理，下面列举了反向代理的具体功能：

- 根据 **HTTP** 请求的内容来控制系统的行为（与七层交换机的作用类似）
- 优化系统整体的内存使用率
- 缓存 **Web** 服务器的应答数据
- 使用 **Apache** 模块控制处理规则

以下依次进行讲解。

2.1.2　根据 **HTTP** 请求的内容来控制系统的行为

因为 IPVS 位于第四层，所以无法根据客户端所需要的 HTTP 请求的内容进行处理的分配。而如果在此引入反向代理，那么根据 HTTP 请求内的网址信息：

- 若客户端请求的网址为 **/images/logo.jpg**，则分配到图片专用的服务器上
- 若客户端请求的网址存在 **/news**，则分配到生成动态内容的 **Web** 服务器上

即可将处理分配到不同的服务器上进行（图 2.1.2）。

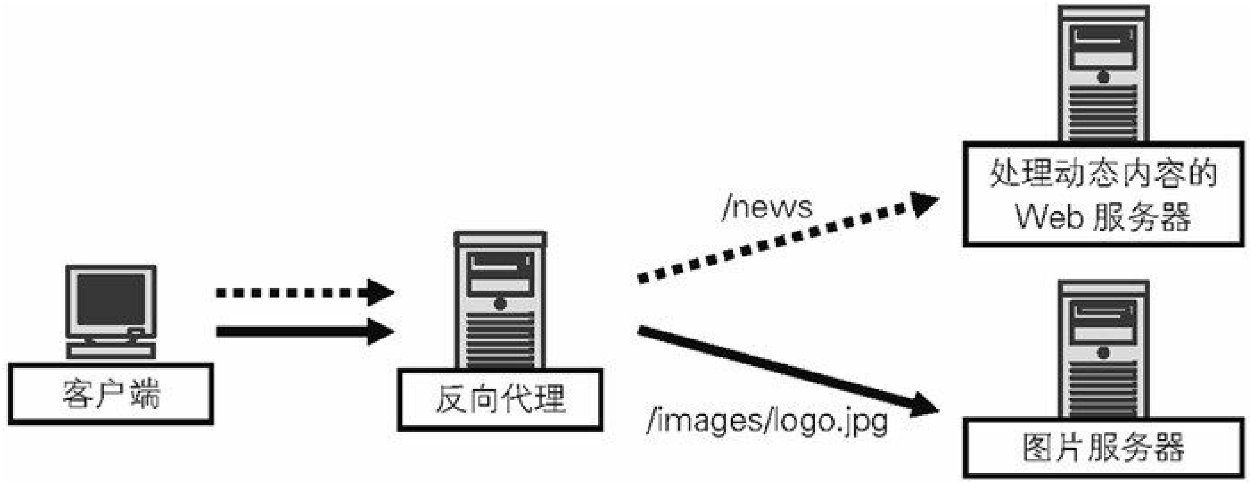


图 **2.1.2**　反向代理

使用 Apache 搭建反向代理的情况下，处理的分配使用的是 mod_rewrite 的 RewriteRule 功能。通过使用 mod_rewrite 进行控制，可以获得很多强大的功能，例如：

- 查看客户端 **IP** 地址，仅允许特定的 **IP** 地址访问服务器
- 通过查看客户端的 **User-Agent**（用户代理），**Web** 服务器可根据客户端的 **User-Agent** 对客户端返回合适的数据
- 将 **/hoge/foo/bar** 修改为 **/hoge?foo=bar**，再请求 **Web** 服务器

以上功能应该在哪里使用呢？以下将进行说明。

根据 **IP** 地址进行控制

通过 IP 地址进行控制可以屏蔽来自恶意主机的请求。此外，在包含面向管理人员的页面的网站中，也可同时通过 IP 地址及网址进行控制，以限制管理人员的页面仅能通过特定的 IP 进行访问。

根据 **User-Agent** 进行控制

User-Agent 的控制是为了应对 Googlebot、Yahoo! Slurp 等搜索引擎机器人（Web Spider，也称蜘蛛）而存在的。

比如，假设在面向用户的网页中存在难以缓存的动态页面（根据用户显

示用户名的页面等），而面向搜索引擎机器人则没有必要显示这些内容，那该页面就可以被缓存。通过查看 User-Agent 的信息，若是搜索引擎机器人的 User-Agent，则可以直接经由缓存服务器来处理该请求。

网址的重写

为了使网站整体的层级结构更清晰，我们有必要使网址让用户看起来更简洁明了。原本客户端的网页浏览器需要支持动态网页，实现“酷网址”[1] 后，在不支持动态网页的旧版浏览器中也可浏览该网页。这是因为在反向代理中，对该网址进行了分解重写，使得旧版浏览器能够识别这些伪静态（将动态网址重写为静态）的网址。在旧版浏览器提交此类链接时，将伪静态的网址请求首先提交给反向代理，由反向代理将伪静态的网址修改为动态网址后，再发给动态服务器进行处理。

> [1] 例如比起 http://b.hatena.ne.jp/bookmark.cgi?user=naoya&tag1=perl&tag=2=cpan 这样的网址，http://b.hatena.ne.jp/naoya/perl/cpan 这样的网址更加简洁整齐、更酷，所以称为酷网址。详情请参考以下链接。
> ***URL*** http://www.w3.org/Provider/Style/URI.html

2.1.3 优化系统整体的内存使用率

在返回动态内容的 Web 服务器中（一般称其为 AP 服务器），通常会将应用程序常驻在内存中，以避免在启动应用程序时产生间接性能成本（Overhead）。例如用 Java 编写的程序，初次启动确实很花时间，若初次启动就常驻内存，以后即可省去启动浪费的时间。在 Perl 或 PHP 中，通过将 mod_perl、mod_php 模块部署到 Web 服务器，即可加快应用软件的速度，这与上述常驻内存的原理相同。在 FastCGI 中，也大致使用了同样的原理实现应用的加速。

在 AP 服务器加速时，需要大量的内存空间。与只返回静态内容的 Web 服务器相比，返回动态内容的 AP 服务器通常需要消耗数倍甚至数十倍的内存空间。

一般 AP 服务器在处理客户端的单个请求时，会将其分配到单个进程或单个线程进行处理。在进程 / 线程中，各进程 / 线程之间都是独立运行的。这样应用软件的开发人员在开发时就无需考虑资源冲突的问题，应用软件的设计也就更简单了。

然而另一方面，在 AP 服务器对单个请求使用单个进程 / 线程来应答时，若需返回图片及 JavaScript、CSS 等静态内容，即原样返回文件中的内容，可以将此类文件分配到静态服务器进行处理，以降低 AP 服务器的负载。

例：动态页面中的请求详情

假设在一个动态生成的 HTML 页面中使用了30张图片。就像“Hatena”网站的主页 [2] 一样。该页面是动态生成的。

> [2] ***URL*** http://www.hatena.ne.jp/

在对该页面的请求中，只有初次请求动态生成了 HTML 页面，客户端的浏览器下载了该页面。浏览器将此 HTML 解析后，向服务器请求了需要的图片及脚本文件，共计 1 个动态请求 +30 个静态请求。

- **——**所有内容均由 **AP** 服务器应答的情况

  1+30 个请求全由 AP 服务器进行应答的情况下，虽然大部分都是返回静态的内容，但为了处理这唯一一个动态的请求，其余 30 个静态请求的应答也会受到拖累，消耗大量的内存。这是因为图片及动态内容的每个请求都需要一个进程 / 线程进行应答（图 2.1.3）。

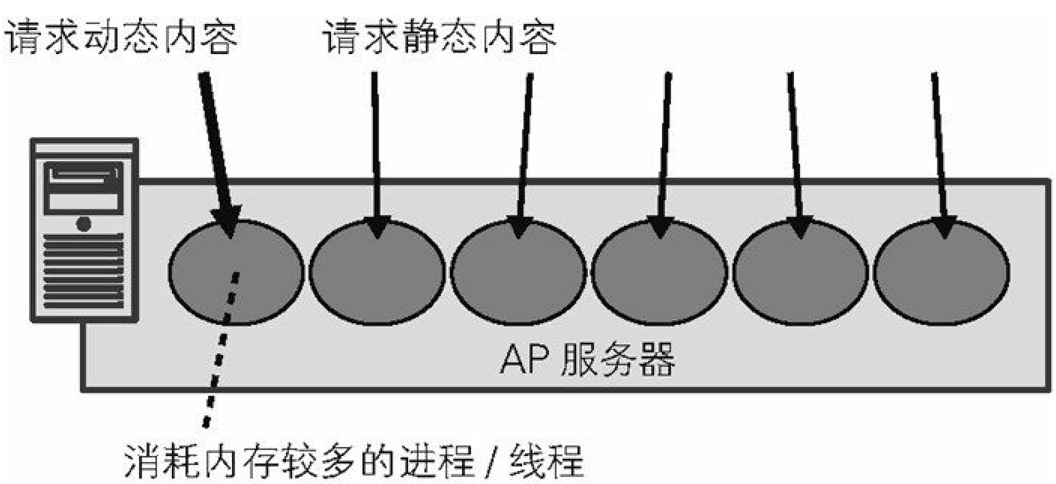


图 **2.1.3**　所有的内容均由 **AP** 服务器应答的情况

- **——**分配到不同服务器的情况

在此将处理分配到不同的服务器上进行，用 Web 服务器返回静态内容，用 AP 服务器生成动态内容（图 2.1.4）。这样一来，静态内容就可以由仅消耗少量内存的 Web 服务器来应答，AP 服务器则只对应用程序的动态内容应答，不仅提高了系统整体的内存使用率，也提高了能够并发处理的请求数。

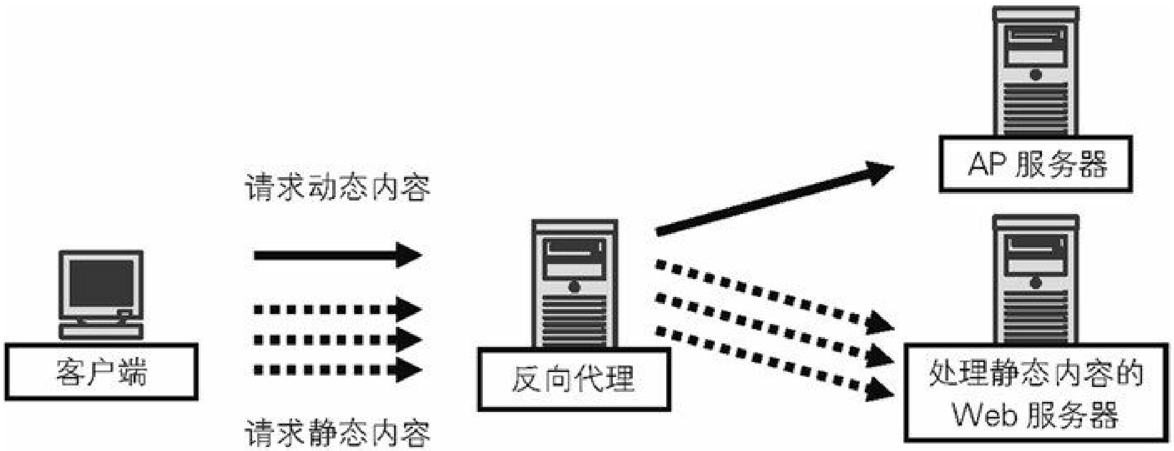


图 **2.1.4**　分配到两台服务器的情况

交由两台服务器分别处理是个很好的选择，但如何将静态内容、动态内容分离到不同的服务器呢？这里就该请出反向代理 了。

- 将 **images** 目录下及 **CSS** 等静态内容的网址交由 **Web** 服务器

- 将除此以外的动态内容的网址交由 **AP** 服务器

即针对网址的内容修改分配的目的地。可以看出反向代理的行为就相当于七层交换机的处理。

常常我们也会将反向代理作为静态服务器来使用（由反向代理直接返回静态内容，无需准备其他的服务器），就像图 2.1.5 那样将静态内容交由反向代理自身直接返回。



图 **2.1.5**　一般的结构

2.1.4　缓存 Web 服务器的应答数据

反向代理分离出了 AP 服务器，为 AP 服务器提供了一个缓冲，这是非常关键的一点。特别是要使用 HTTP 的 Keep-Alive 功能时，反向代理的存在就显得尤为重要。

HTTP 的 Keep-Alive

HTTP 中有一个 **Keep-Alive** 功能，在某个客户端一次性将多个内容从同一台服务器读取时，经常会用到该功能。以之前举出的显示了 30 个图片的 HTTP 页面为例，通常每个 HTTP 请求都需要与服务器重复进行连接、切断、连接……的操作，处理性能并不高。若在第一次请求完成后保持连接不切断，当再次请求时，直接使用以前建立过的连接，即可实现用一次连接来处理更多的请求。

要实现该功能就需要使用 Keep-Alive。只需在服务器中设定好 Keep-Alive，浏览器即可根据 HTTP 头判断是否启用 Keep-Alive。启用时，浏览器依照 Keep-Alive 的规定保持与服务器的连接，只需一个连接就可实现多个文件的下载。事实上与关闭 Keep-Alive 的服务器相比，开启该功能的服务器的下载速度明显会快一些。

使用 Keep-Alive 保持连接时，理论上会加重 Web 服务器的负载。具体来说，从收到某个特定的客户端的请求开始，在一定时间内，进程 / 线程会被占用以应答客户端的请求 [3] 。

> [3] lighttpd 等采用事件模式的 Web 服务器则并不一定如此。

例：内存消耗与 **Keep-Alive** 的开启 **/** 关闭

首先从内存消耗的角度来考虑该情况。在每个进程消耗较多内存的 AP 服务器中，一台主机能启动的最大进程数仅有 50 ～ 100 个左右。若不使用反向代理而开启 Keep-Alive，那么在这为数不多的 50 ～ 100 个进程中，大部分都会为保持 Keep-Alive 的连接而被消耗掉（图 2.1.6）。

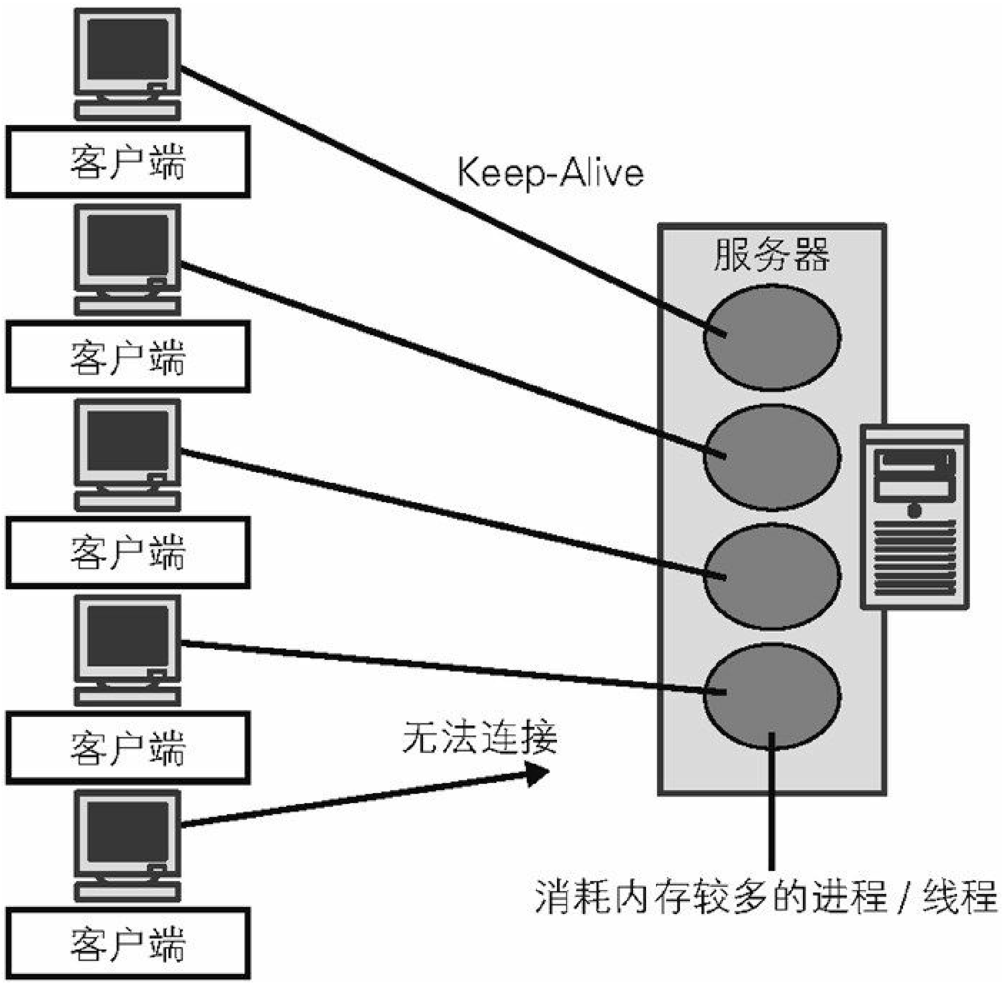


图 **2.1.6**　为保持 **Keep-Alive** 的连接而使进程被消耗

若 Keep-Alive 关闭，从客户端就可明显感觉到浏览速度变慢。这不是我们想要的结果。

下面考虑一下增设反向代理的情况。通常作为反向代理工作的 Web 服务器中，由于每个进程消耗的内存并不多，一台主机可以启动 1,000 ～ 10,000 个进程。这种情况下，即便 Keep-Alive 为了保持连接而消耗一定程度的进程，也不会增大负载。然后，只在客户端与反向代理间开启 Keep-Alive，反向代理与后端的 AP 服务器间则关闭 Keep-Alive（图 2.1.7）。

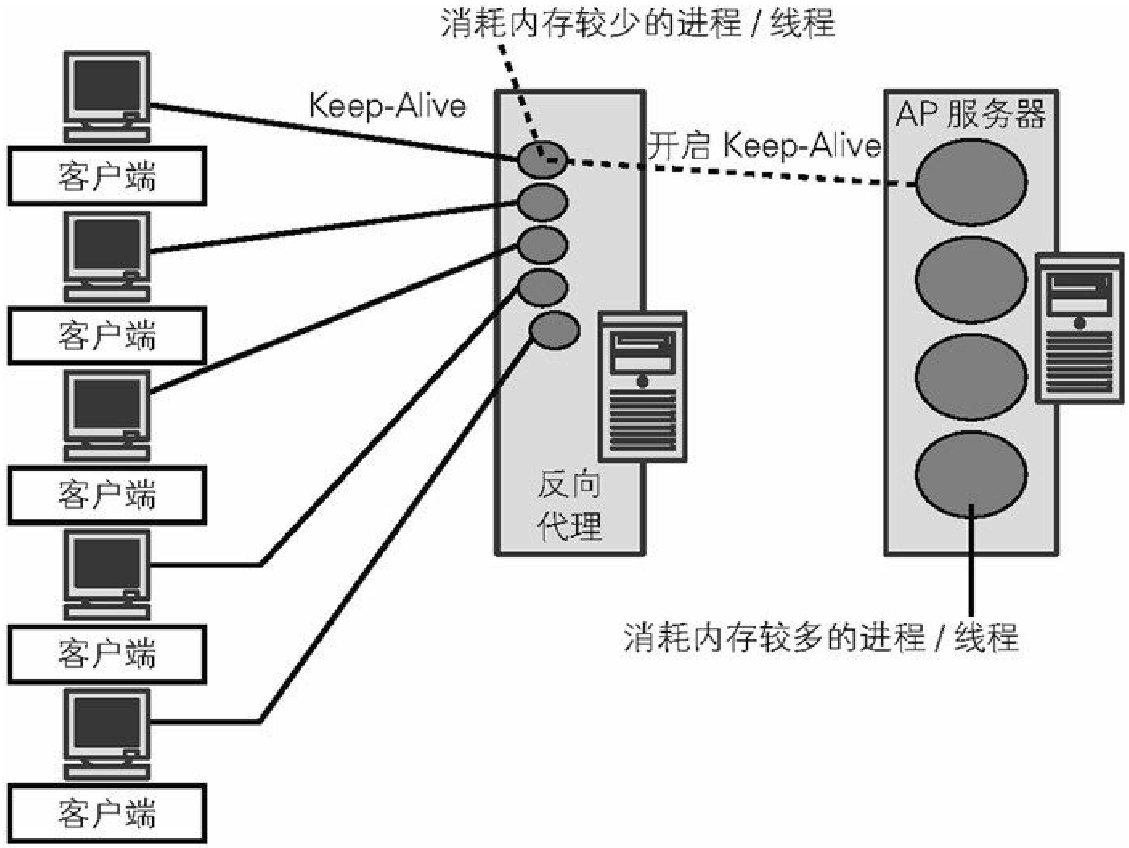


图 **2.1.7　Keep-Alive** 的开启 **/** 关闭

这样即使 AP 服务器的进程数较少，在一个请求结束后也可立刻应答别的请求。整体看来能够并发处理的请求数有所增加，吞吐量的承载能力也显著提高了。因为反向代理承担了与客户端保持连接的任务，而消耗内存较多的 AP 服务器则不用承担该任务，照此即可搭建良好的服务器拓扑结构。

2.1.5　使用 **Apache** 模块控制处理规则

在反向代理使用 Apache 搭建的情况下，该反向代理可以通过利用 Apache 的模块，在程序中控制 HTTP 请求的前端 / 后端的行为。

例如，Apache2.2 的源码中附带了 **mod_deflate** 模块，该模块可将网页内容使用 gzip 压缩。若在反向代理中使用此模块，就可以将后端的 AP 服务器返回的 HTTP 应答进行压缩，然后再通过反向代理返回客户端（图 2.1.8）。同样，若使用 **mod_ssl** 模块，还可将 AP 服务器返回的应答进行 SSL 加密处理。

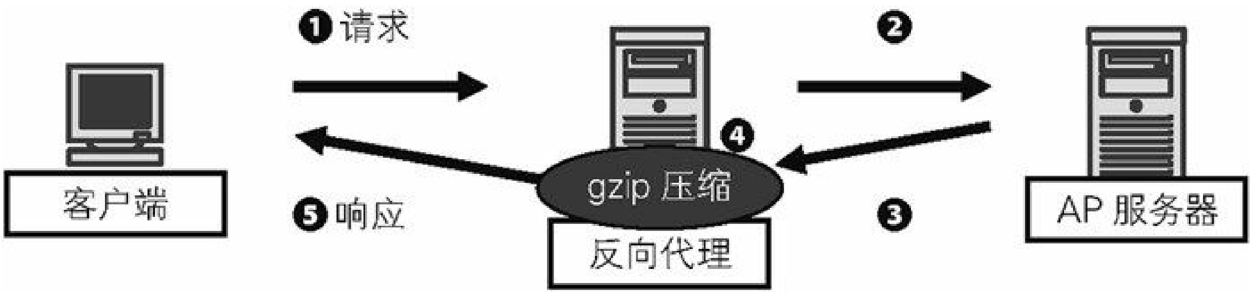


图 **2.1.8**　使用 **mod_deflate** 模块的情况

另外，Apache2.2 中还有用来应对 DoS 攻击的 **mod_dosdetector** 模块 [4]，该模块可在某个客户端发出过多请求时，临时隔离这些请求。若将其合理设置在反向代理中，即可防止后端的 AP 服务器因承受过多的请求而超出负载。

> [4] ***URL*** http://sourceforge.net/projects/moddosdetector/

除 Apache 外，lighttpd 等第三方的模块及插件也有很多，在反向代理中使用这些功能，同样会为生产环境增光添彩。

增设反向代理的判断

在使用 AP 服务器发布动态内容的情况下，反向代理的存在与否会对系统的灵活性产生很大的影响。即使只有一台主机，通过在该主机内同时运行反向代理与 AP 服务器，将“静态内容的发布工作与后端 AP 服务器”分离进行，也可提高资源的使用效率。

所以真的找不到不增设反向代理的理由啊！

2.1.6　增设反向代理

以下对使用 Apache 搭建反向代理的方法，以及所搭建的反向代理的各种设定进行讲解。

使用 **Apache 2.2**

搭建反向代理最好使用稳定版的 Apache 2.2[5]。另外，鉴于 prefork 会为每个客户端分配一个进程，为了让反向代理尽可能地承受更多的并发量，建议使用“worker 模式”，因为该模式只需为每个客户端分配一个线

程即可。

> 5 本节中使用了 CentOS 4.4 和 Apache 2.2.4。

以 **worker** 模式启动 **httpd**

在 Red Hat Enterprise Linux 5 及 CentOS 5 的标准软件包内默认附带了 Apache 2.2，一般无需另外安装。在 Rad Hat 中可以选择服务器启动时运行的软件包模式，只需在 /etc/sysconfig/httpd 中加入一行：

```
HTTPD=/usr/sbin/httpd.worker
```

即可实现以 worker 模式启动 httpd。

httpd.conf 的配置

以下讲解将 Apache 作为反向代理运行所需要进行的准备工作。Apache 服务器的一个重要特性就是其模块化的结构，即 **DSO** （Dynamic Shared Object，动态共享对象）的特性。

设定最大进程 / 线程数

首先设定 worker 的进程和线程数。

```
StartServers 2
MaxClients 150
MinSpareThreads 25
MaxSpareThreads 75
ThreadsPerChild 25
MaxRequestsPerChild 0
```

默认的 httpd.conf 中全局指令（Global Directives）的配置如上所示，该设定参数均为保守值。因为我们希望反向代理能够承受较高的负载以保护后端服务器，因此可以多使用些系统资源，以确保反向代理能够同时

处理足够多的请求数。

以上需要关注的参数为 **MaxClients** 及 **ThreadsPerChild** 。使用 worker 模式的情况下，Apache 会启动多个子进程，并在各个子进程中生成多个线程，这样一来就能处理可观的请求量了。

此处的 ThreadsPerChild 控制各个进程中的线程数，这里子进程的最大值（即子进程的数量）就是：

```
MaxClients ÷ ThreadsPerChild
```

MaxClients 决定了能够同时处理的客户端访问总数。例如在前文那样的设定的情况下，即为

- 最大进程数：**6**
- 单位进程的最大线程数：**25**
- 所支持的客户端并发数：**6x25 = 150**

若安装有 2GB ～ 4GB 的内存，即可承受 1,000 ～ 10,000 左右的并发量。例如，

- 最大进程数：**32**
- 单位进程的最大线程数：**128**
- 所支持的客户端并发数：**32x128 = 4096**

若需实现以上情况，可按照以下的具体参数进行设定：

```
StartServers          2
ServerLimit          32 ←新增
ThreadLimit         128 ←新增
MaxClients         4096 ←修改
MinSpareThreads      25
MaxSpareThreads      75
```

```
ThreadsPerChild     128 ←修改
MaxRequestsPerChild   0
```

可以看出新增了 **ServerLimit** 及 **ThreadLimit** 这两个项目。ServerLimit/ThreadLimit 的设定决定了最大进程 / 线程数，MaxClients 及 ThreadsPerChild 是一组与 ServerLimit/ThreadLimit 相对应的设定项目。因为 ServerLimit 的默认值是 16，ThreadLimit 的默认值是 64，所以在需要实现比该默认值高的进程 / 线程数的情况下，必须明确指定这两个项目。

- **——ServerLimit/ThreadLimit** 与内存的关系

  然而，既然已经有 MaxClients 及 ThreadsPerChild 决定最大进程 / 线程数，那为什么还要设置另一组参数 ServerLimit 及 ThreadLimit 呢？这是因为 MaxClients 及 ThreadsPerChild 是服务器方有关动态资源消耗的设定项目，该值的高低对服务器的资源消耗量没有影响。而 ServerLimit 及 ThreadLimit 则会影响 Apache 所确保的共享内存的大小。若将这些项目的设定值设定得比必要值还高，就会导致内存空间的浪费。因此，若进程数 / 线程数的目标设定值超过了 ServerLimit 16 及 ThreadLimit 64，就有必要仔细斟酌以确保合理设定。

  进程 / 线程的内存使用量与启动的模块种类有关。由于系统分配内存给 Apache 时需要看具体环境，因此无法决定该设定值。此处的配置仅做参考。至于在实际使用的环境中如何设定上限值，还需具体评估进程 / 线程的内存使用量。关于这部分内容，我们将在第 4 章进行详细讲解。

  设定某个 Web 服务器的最大进程 / 线程数时，应该确保该主机在达到访问上限时不会启用 Linux 的 SWAP 交换区，即：

  - 系统或 **Web** 服务器以外的软件常驻所使用的内存量
  - **Web** 服务器的进程 / 线程数达到上限时，服务器消耗的总内存量

计算以上两者的总和，并兼顾实际可使用的内存总量进行设定。关于识别单位进程 / 线程的内存使用量的方法，将在第 4 章详细讲解。

Keep-Alive 的配置

如前所述，开启反向代理的 Keep-Alive 并关闭 AP 服务器的 Keep-Alive 是关键。对全局指令做出如下设定即可开启 Apache 的 Keep-Alive。

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 5
```

以上设定指示了下面的内容。

- 开启 **Keep-Alive**
- 指定 **Keep-Alive** 所处理的最大请求数为 **100**
- 指定 **Keep-Alive** 的超时数（**KeepAliveTimeout**，与客户端保持连接的时间）为 **5** 秒*

KeepAlive Timeout 的默认时间为 15 秒。在一般的 Web 网站中，应该可以在 5 秒内返回响应，若该值偏大，仅进程 / 线程的 Keep-Alive 就会占用很多时间，导致服务器的资源被大量消耗。所以将默认值设定为比 15 秒更小的数字不会有什么问题。

载入必要的模块

接下来应该配置的是载入必要的模块。搭建反向代理所需的基本的模块有下面几种：

- **mod_rewrite**
- **mod_proxy**
- **mod_proxy_http**

添加以上模块时，还需载入“mod_alias”模块。

```
LoadModule alias_module modules/mod_alias.so
LoadModule rewrite_module modules/mod_rewrite.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

将这些模块载入后，即可使用 RewriteRule 及 RewriteRule 中的 Proxy、Alias 等指令。

默认情况下，Apache 安装包的 httpd.conf 中载入了很多模块。由于载入不必要的模块会加大 Apache 的内存消耗量，因此请尽量减少不必要的模块的加载。

设定 **RewriteRule**

在完成 ServerRoot 及权限日志等的设定后[6]，最后就要设定 RewriteRule 了，这是搭建反向代理的核心。

> [6] 关于 Apache 的基本设定，请参考 Apache 的使用手册。

这里考虑进行以下设定：

- 当访问 **/image** 目录下的图片时，此类网址由反向代理自身处理。所有图片都在反向代理主机内的 **/path/to/images/** 目录下存放
- 对 **/css**、**/js** 也做以上处理
- 除此以外的动态内容的处理，请求 **AP** 服务器 **192.168.0.100** 进行代理

该设定如代码清单 2.1.1 所示。

代码清单 **2.1.1**　**RewriteRule** 等的设定

```
Listen 80
<VirtualHost *:80>
```

```
  ServerName naoya.hatena.ne.jp

  Alias /images/ "/path/to/images/"
  Alias /css/    "/path/to/css/"
  Alias /js/     "/path/to/js/"

  RewriteEngine on
  RewriteRule ^/(images|css|js)/ - [L] ←❶
  RewriteRule ^/(.*)$ http://192.168.0.100/$1 [P,L] ←❷
</VirtualHost>
```

请看 RewriteRule 的内容。RewriteRule 将请求的网址进行模式匹配（Pattern Match），若匹配成功，则将指示其做出相应的处理。RewriteRule 可以通过正则表达式定义。在代码清单 2.1.1 中， ❶ 处的配置原则是：

无论遇到任何包含 **/images/**、**/css/**、**/js/** 的 **URL**，都执行匹配的动作（**[L]** 是 **RewriteRule** 的匹配模式在此结束的意思），默认通过内容处理器（**ContentHandler**）返回内容

从默认的内容处理器返回静态文件，根据 URL 查找响应内容并返回到客户端，这是 Apache 的一贯动作。

根据该配置，当客户端发出请求时，若该请求为 /images/profile/naoya.png，本地的 /path/to/images/profile/naoya.png 就会被返回到客户端。

根据以上内容做出代码清单 2.1.1 的 ❷ 处的设定，即：

对全部 **URL** 的请求均由 **192.168.0.100** 进行代理

如此便设定完毕。这时如果通过浏览器访问反向代理所监听的端口，就会显示 192.168.0.100 返回的内容，就像是 AP 服务器做出了应答。

2.1.7 进一步对 RewriteRule 进行设置

以下逐步介绍 RewriteRule 的设定例子。

禁止来自特定主机的请求

例如，禁止来自特定 IP 地址的请求，使其返回 403 错误代码的设定如代码清单 2.1.2 所示。

代码清单 **2.1.2**　设定示例 **1**

```
RewriteEngine on

# 若是192.168.0.200的请求，则向客户端返回403
RewriteCond %{REMOTE_ADDR} ^192\.168\.0\.200$
RewriteRule .* - [F,L]

# 反向代理的设定
RewriteRule ^/(images|css|js)/ - [L]
RewriteRule ^/(.*)$ http://192.168.0.100/$1 [P,L]
```

在代理 AP 服务器之前，通过进行条件判断的 RewriteCond 查看 REMOTE_ADDR，即可对特定地址返回 403 信息 [7] 。

> [7] 使用 RewriteRule 的标志 [F,L]。

对于来自搜索引擎机器人的请求使用缓存服务器

假设使用 Squid 等架设的 HTTP 缓存服务器被配置在 192.168.0.150。若想要针对来自搜索引擎机器人的请求进行特殊配置，也就是对特定 User-Agent 的请求直接返回已经缓存的内容，可以如代码清单 2.1.3 那样进行设定。

代码清单 **2.1.3**　设定示例 **2**

```
# 为了使SetEnvIf指令生效，需要读取mod_setenvif
LoadModule setenvif_module modules/mod_setenvif.so

# 若User-Agent中包括“Yahoo! Slurp”或“Googlebot”
# 则环境变量IsRobot为真
SetEnvIf User-Agent "Yahoo! Slurp" IsRobot
SetEnvIf User-Agent "Googlebot" IsRobot
```

```
RewriteEngine on

# 若环境变量为真，则代理缓存服务器
RewriteCond %{ENV:IsRobot} .+
RewriteRule ^/(.*)$ http://192.168.0.150/$1 [P,L]

# 除此以外的情况下按照如下方式处理
RewriteRule ^/(images|css|js)/ - [L]
RewriteRule ^/(.*)$ http://192.168.0.100/$1 [P,L]
```

使用 mod_setenvif 根据 User-Agent 的文本判断是否为来自搜索引擎机器人的访问，若判定确实为搜索引擎机器人，则代理到缓存服务器。

Apache 的 mod_rewrite 可与其他的模块等配合使用，实现弹性化的强大设定。因此只要满足 RewriteRule 描述的条件，就总能分配到其他服务器。

2.1.8 使用 mod_proxy_balancer 向多台主机分流

在此讨论当有多台后端 AP 服务器的情况下该采用哪种拓扑结构。下面有几个方案可供思考：

❶ 反向代理与 **AP** 服务器总是成对配置。将请求从特定代理传递到特定 **AP** 服务器

❷ 使用 **mod_proxy_balancer**，将请求从特定反向代理分配到多个 **AP** 服务器（图 **2.1.9**）

❸ 在反向代理与 **AP** 服务器之间插入 **LVS**

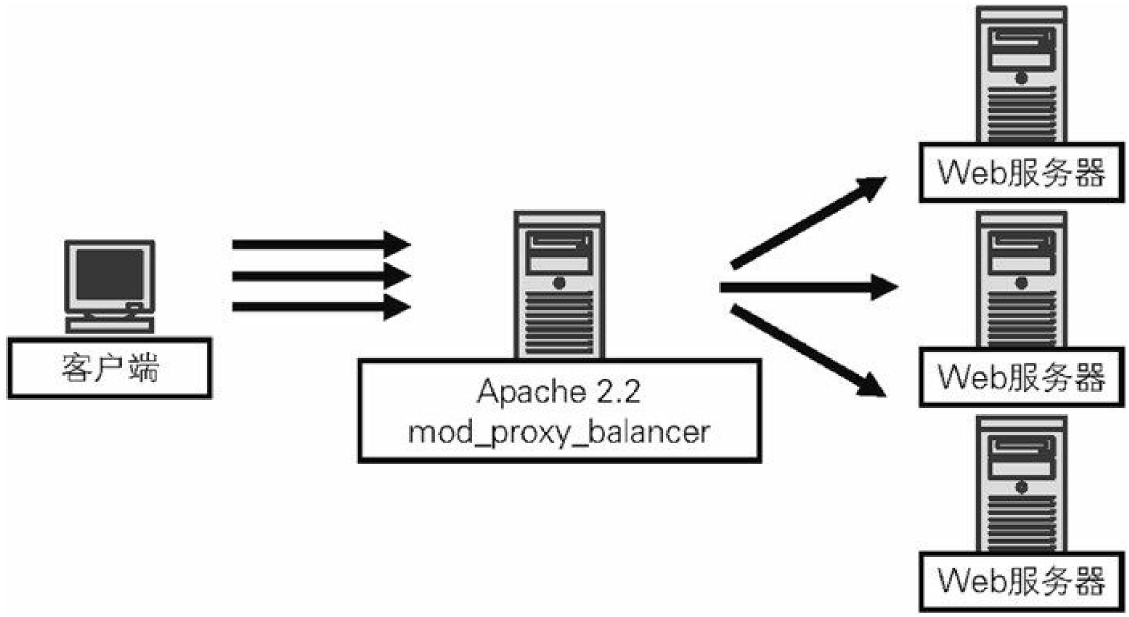


图 **2.1.9**　使用 **mod_proxy_balancer**

在上述内容中， ❶ 不能说是明智的选择。反向代理肯定比 AP 服务器的资源消耗少，而且也需要它来承担那些即使资源消耗小也能正常进行的工作。因此在通常的系统中，出于对冗余的考量，设置两台反向代理已经足够。但受负载情况的影响，很多情况下仅有两台后端 AP 服务器是不够的。例如在本书执笔时，Hatena 书签 [8] 就是由两台反向代理与 11 台 AP 服务器组成的拓扑结构。

[8] ***URL*** http://b.hatena.ne.jp/

由于反向代理与 AP 服务器的资源消耗不同，若是成对配置，反向代理方面势必会造成无端的资源浪费。

❷ 中 Apache 的 mod_proxy_balancer 这一模块的作用是，在进行反向代理时，将请求通过某种算法分配给代理目标的多台主机。如果代理的目标主机没有办法返回应答，则会启动故障转移，使请求不会代理给该主机，即可实现代理到后端的请求总是会分配到正常工作的主机。利用好该模块的这点特征是一个方法。

另一个方法是，在反向代理与 AP 服务器之间插入 LVS+keepalived，即 ❸ 。这确实是很实用的方法，根据笔者个人的使用感受，LVS 的故障转移功能并不输于 mod_proxy_balancer。另外在 LVS+keepalived 与

mod_proxy_balancer 的较量中，LVS+keepalived 的情况下负载分流的逻辑更容易调整，而且也可以通过命令行进行管理，相当方便。

但 LVS+keepalived 需要事先增加若干个服务器。以下以想要进行简易的负载分流为例，讲解 mod_proxy_balancer 的使用方法。

mod_proxy_balancer 的使用示例

使用 mod_proxy_balancer 搭建反向代理是很简单的。

- 加载 **mod_proxy_balancer** 模块 [9]
- 在 **BalancerMember** 指令中定义分流目的地的主机一览表
- 对反向代理进行 **RewriteRule** 设定（规则重写设定）。在此可使用 **balancer://Scheme** 的形式

> [9] mod_proxy_balancer 是 Apache 2.2 中的标准功能。

假设 AP 服务器有 3 台，IP 地址是 192.168.0.100 ～ 102。这里的 httpd.conf 的设定如代码清单 2.1.4 所示。

代码清单 **2.1.4**　通过 **mod_proxy_balancer** 搭建反向代理的设定示例

```
# 加载mod_proxy_balancer模块
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so

# 定义分流目的地的主机一览表
<Proxy balancer://backend>
  BalancerMember http://192.168.0.100 loadfactor=10
  BalancerMember http://192.168.0.101 loadfactor=10
  BalancerMember http://192.168.0.102 loadfactor=10
</Proxy>

Listen 80
<VirtualHost *:80>
  ServerName naoya.hatena.ne.jp

  Alias /images/ "/path/to/images/"
  Alias /css/    "/path/to/css/"
  Alias /js/     "/path/to/js/"
```

```
  # 设定反向代理
  RewriteEngine on
  RewriteRule ^/(images|css|js)/ - [L]
  RewriteRule ^/(.*)$ balancer://backend/$1 [P,L] ←❶
</VirtualHost>
```

请看代码清单 2.1.4。之前的例子中都是直接写 http://192.168.0.100/$1 与 AP 服务器的网址，而本次则使用“`balancer://Scheme`”这种形式。当表述为 `balancer://backend` 时，每个请求都会选择上层中定义的 BalancerMember 中的一台。至于具体为哪个服务器，则与 BalancerMember 中定义的 loadfactor 的值有关。loadfactor 的值越大，则被分配的概率越高。像代码清单 2.1.4 那样，设定 loadfactor 均为同一值的话，即可获得均等的分配效果。

mod_proxy_balancer 中还存在除 loadfactor 以外的设定值。在实际使用中，应该根据网站的结构设定合适的数值。

2.2　增设缓存服务器 ——Squid、memcached

2.2.1　引入缓存服务器

在 2.1 节中已经针对反向代理进行了讲解。接下来，本节将要讨论缓存服务器的详情。

HTTP 与缓存

在网络服务所使用的协议中，HTTP 是稀疏的协议，即具有无状态 [10] 的属性。因为由一个无状态的协议所交换的文件不具备状态，因此具有易缓存的特性。所以在 HTTP 协议中，提供了非常强大的缓存机制。

> [10] 关于 HTTP 无状态协议，请参考以下链接。
> ***URL*** http://baike.baidu.com/view/4551466.htm

例如在 Internet Explorer（IE）或 Firefox 等大多数 Web 浏览器中，当初次访问下载该文件时，浏览器会将其缓存在本地，当再次访问时，将直接从本地调出该文件。在浏览器中，为了查看远程文件是否已经更新，

可以使用 HTTP 头替换服务器与文档的修改日期。

通过 **Live HTTP Headers** 得知缓存效果

通过使用 Fierfox 浏览器的 **Live HTTP Headers** [11] 功能，能够了解到 HTTP 头的交换情况。下面以图片文件的交换情况为例进行说明。以下为浏览器向 Web 服务器发送的 HTTP 请求头。

> [11] ***URL*** https://addons.mozilla.org/zh-CN/?refox/addon/live-http-headers/

```
GET /images/top/h1.gif HTTP/1.1
Host: www.hatena.ne.jp
Keep-Alive: 300
Connection: keep-alive
If-Modified-Since: Wed, 19 Dec 2007 15:31:43 GMT
```

If-Modified-Since 头中记载了日期。这是浏览器取得该文件的时间。该请求所得到的服务器（Apache）的应答为：

```
HTTP/1.x 304 Not Modified
Date: Wed, 27 Feb 2008 06:43:31 GMT
Server: Apache
```

由此可以看出 HTTP 的状态代码为 304（Not Modified）。Web 服务器 Apache 所进行的处理为：

- 取得从客户端发来的 **If-Modified-Since** 的更新时间
- 与本地文件的时间进行比较
- 判断客户端所保存的缓存文件是否需要进行更新

在此并不会返回文件，而是返回状态码 304，即代表“即使不返回文件，只要本地存在该文件的缓存，就能显示出图片”。据此，

- 客户端可以省略从网上下载图片数据的步骤
- 服务器可以省略将文件传送给客户端的步骤

以上就是使用 HTTP 协议的缓存所达到的效果。

2.2.2　Squid 缓存服务器

既然客户端与服务器有 HTTP 缓存，服务器与服务器间应该也有类似 HTTP 缓存的方法存在。不管主机与主机间的关系究竟如何，若两者的交换中有 HTTP 协议可用，那应该就能实现内容的缓存。

Squid [12] 是 HTTP、HTTPS、FTP 等使用的开源缓存服务器。Squid 可用于任意 Web 系统中，使其实现 HTTP 的缓存功能。将 Squid 配置在使用 HTTP 进行通信的两点之间，即可实现缓存。

> [12] ***URL*** http://www.squid-cache.org/

Squid 通常被用来缓存客户端从服务器上下载的文件。例如大学或企业在 LAN 网关配备了 Squid，当办公室的各 PC 访问 Internet 的网站时，都会经由 Squid，即缓存服务器 Squid 发挥了代理服务器的功能（图 2.2.1）。

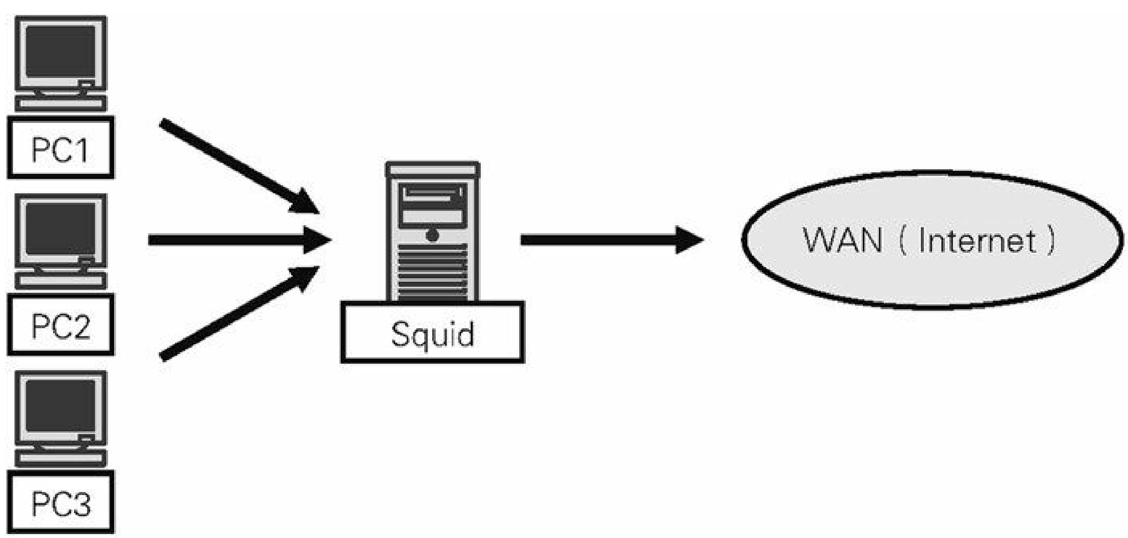


图 **2.2.1　Squid**（代理服务器）

在此，当某个客户端下载了文件，Squid 就会缓存它。当其他的客户端请求同样的文件时，将从缓存中直接取出该文件。由于文件一旦被请求后，之后就会直接从缓存中返回，因此无需每次都请求原始网页，不仅节约了网络带宽，因为从 LAN 直接返回数据，下载文件的速度也会变得很快。

使用 **Squid** 搭建反向代理

Squid 可作为反向代理使用。在 2.1 节中介绍了使用 Apache 搭建反向代理的步骤，Squid 也可通过类似的步骤搭建起反向代理。从服务器负载分流的观点来看，这才是 Squid 主要的应用形式。

Squid 作为反向代理工作时，通过 Squid 可以将网站的文件缓存到 Squid 服务器上（图 2.2.2）。

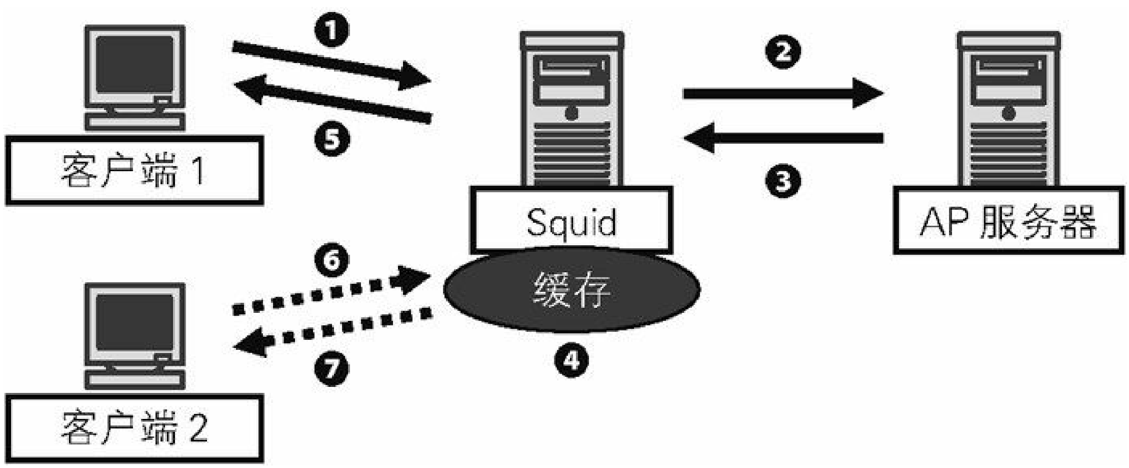


图 **2.2.2**　**Squid**（反向代理）

- 若 **Squid** 接到客户端的 **HTTP** 请求，就向后端服务器索取请求的文件
- **Squid** 将服务器取得的文件缓存到本地
- 当其他客户端发出请求时，**Squid** 将确认缓存的有效性，若缓存有效，就将文件从缓存直接返回到客户端
- 如果短时间内有 **10,000** 个客户端请求同一文件，则后端服务器（若文件可被缓存）只对第一次请求做出响应，其余 **9,999** 个请

求均由 **Squid** 的缓存返回

Squid 内部的存储器专为缓存设计，速度非常理想，而且 Squid 可以以较小的资源消耗处理大规模的请求。大多数情况下，与从后端服务器返回文件相比，从 Squid 的缓存中返回文件速度很快。这样做也会降低负载。

Squid 不仅仅只是缓存 HTTP 的内容，也可以通过网络和其他的 Squid 服务器共享缓存。使用该功能时，返回缓存的 Squid 服务器在负载升高的情况下，只需将请求导向其他的 Squid 处理即可。这与冗余的实现是类似的。

Squid 缓存什么

Squid 是以 HTTP 协议的缓存功能为前提的缓存服务器。因此缓存 HTTP 文件、CSS 样式表、JavaScript 脚本、图片等静态内容是很明智的选择。若原始文档更新的话，Squid 缓存会将过期的缓存删除并替换为最近的内容。

动态的内容要如何处理呢？ Squid 拥有非常灵活的缓存命令，例如针对某个动态页面设置其只缓存 30 分钟也没问题。

基于 HTTP 协议进行缓存，也就是把 URL 作为 Key 来缓存内容。即每个 URL 对应一个缓存文件，以此存储缓存数据。

但问题是动态页面包含有状态信息。例如在 Web 应用软件的页面头中很多都会显示账户名等。一般该情况可通过 Cookie 会话管理来实现。最终即使是同一 URL，根据用户的不同所输出的信息也不同。

若不小心将所有这样的动态文件都进行了缓存，如缓存了“欢迎 A 先生”这样的信息，那么当 B 先生访问同一 URL 时，就会取用 A 先生的缓存信息，对 B 先生输出“欢迎 A 先生”这样的信息。这是缓存中会发生的代表性问题（图 2.2.3）。



图 **2.2.3**　不该缓存的内容

动态页面中，每个用户的行为都会造成页面内容的改变，如果要将URL 作为 Key 来缓存这些文件的话，在 HTTP 协议下是很难实现的。首先，在基于 HTTP 协议的缓存中，通常都是无状态的内容交换，而在通常使用的 Cookie 的会话管理中，无状态的协议则会被加入“状态”，即变成有状态的通信。这有悖于协议所要求的前提条件，因此在该协议所支持的缓存机制中就会产生矛盾。

希望使用缓存来减轻负载，但又无法应用 HTTP 协议级别的缓存……在此情况下，可以在应用软件的内部，例如在数据库（Database）中，将记录的对象以对象为单位进行缓存，以这种方式来解决问题的话，缓存内容的粒度要比平时细。由于粒度变细，因此将不选择 Squid 这种以更大粒度为对象的缓存服务器，仅会选择适合该粒度的缓存服务器。后文中提到的“memcached”就是一个例子。

总之，在可以缓存整个页面的情况下，使用 Squid 是比较明智的选择。

Squid 的设定示例

以下对将 Squid 作为反向代理使用时的设定示例进行介绍。

在将 Squid 作为反向代理使用的情况下，可以使用多种拓扑结构，本文选择的结构是将 Squid 插入反向代理（通过 Apache 构建的）与 AP 服务器中间（图 2.2.4）。在到达存有原始内容的后端 AP 服务器之前，反向代理将经由 Apache → Squid 两处。为了实现分流与冗余，需要准备两台 Squid，并共享缓存。

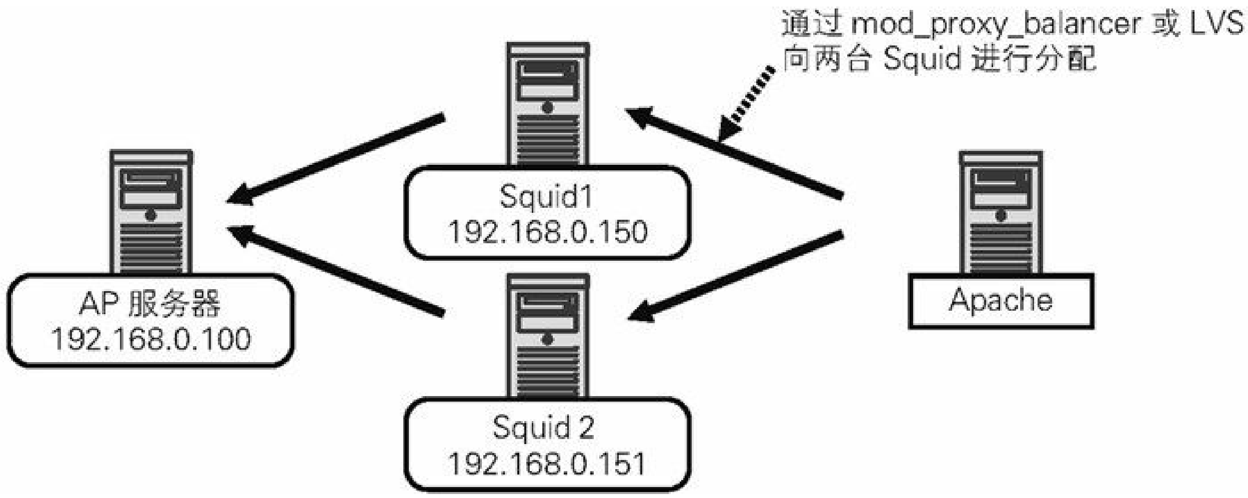


图 **2.2.4**　将 **Squid** 作为反向代理使用的例子

代码清单 2.2.1 是将后端服务器返回的内容缓存 30 分钟，这里的缓冲行为不会识别内容具体是静态还是动态。Squid 的设定文件 **squid.conf** 如代码清单 2.2.1 所示。

代码清单 **2.2.1**　**squid.conf** 的设定示例

```
http_port 80 ←Squid的端口为80      存有原始内容的服务器是后端服务器
cache_peer 192.168.0.100 parent 80 0 no-query originserver ←(192.168.0.100:
cache_peer 192.168.0.151 sibling 80 3130 ←子邻居的(sibling)Squid是192.168.0.

http_access allow all ←从所有的服务器都可访问(由于在局域网内，因此无需限制访问)
cache_dir coss /var/squid/coss 8000 block-size=512 max-size=524288 ←缓存存储

refresh_pattern . 30 20% 3600 ←缓存30分钟 ※2

client_persistent_connections off ←┐
                                   │ ←将Keep-Alive的连接保持设为无效 ※3
server_persistent_connections off ←┘
```

```
icp_query_timeout 2000 ←将确认兄弟缓存是否存在的超时时间设为2000ms
```

※1 coss 是 Squid 的缓存存储器的一种，是现在可使用的存储器中最快的。具体请参考 Squid 手册。

※2 Squid 的缓存控制由 refresh_pattern 进行。由于篇幅原因，这里省略了对 refresh_pattern 的讲解。在 squid.conf 中有详细说明。

※3 关于 Keep-Alive 的无效化，请参考 2.1 节。

2.2.3 使用 **memcached** 进行缓存

Squid 的优点在于可以使用 HTTP 协议将文件进行缓存。HTTP 协议是无状态的可扩展协议，Squid 也同样是可扩展的，而且不受应用软件的结构等的影响。

像之前提到的那样，很多情况都不适合用 HTTP 级别的缓存。在 Web 应用程序的世界中，可以使用缓存服务器，根据应用程序内部使用的数据粒度来管理缓存。**memcached**[13] 就是其中一例。

> [13] ***URL*** http://www.danga.com/memcached/

memcached 是用 C 语言写的应对高速网络分流的缓存服务器，所用的存储器其实是系统的内存。服务器启动 memcached，通过使用专用的客户端库与服务器进行通信，可以取得并保存编程语言中规定的对象。客户端库已经针对不同的语言，发布了与其对应的客户端库，不仅限于 C 语言、C++、Java、Perl、Ruby、PHP、Python，现今流行的语言几乎都获得了支持。

memcached 是由程序内部使用的。在程序内部，虽然经常会将特定数据缓存在文件中，或者缓存在本地内存中，但有时也会希望将其缓存在网络上的服务器中。memcached 就提供了这类问题的解决方案。

在此我们不做过于详尽的讲解，以下仅以简单的 Perl 脚本为例对其使用情况做一个大概的介绍。代码清单 2.2.2 是一个简单的程序，所做的处理仅仅是取得数组对象并将其保存在缓存服务器中。

代码清单 2.2.2 的执行结果是：

```
% perl memd.pl
256
```

代码清单 **2.2.2　memcached** 的使用示例

```
#!/usr/bin/env perl
use strict;
use warnings;

use Cache::Memcached;

## 将192.168.0.1:11211中运行的memcached作为缓存服务器
my $memcached = Cache::Memcached->new({ servers => [ '192.168.0.1:11211' ]

my $object = [ 1, 2, 4, 16, 256 ];

## 将对象通过'object1'保存
$memcached->set( 'object1' => $object );

## 将对象从缓存中获取
my $cached = $memcached->get('object1');

printf "%d\n", $cached->[4];
```

访问从缓存中取出的数组对象，可以确认确实已经准确恢复到了以前的状态。

由于 memcached 是以`(key,value)`键值对进行存储的，不管对象是否为依赖于语言的对象，都能够被保存（进行序列化并保存）。只要知道 key，就可以从别的程序中取得那个缓存。

memcached（根据所装载的客户端库）存在抗障碍性。若某特定的主机运行的 memcached 宕机，客户端库会感知其问题，从而避免将该服务器作为缓存服务器使用。

2.3 MySQL 同步 ——发生故障时的快速恢复

2.3.1 万一数据库服务器停止

大多数情况下，数据库会存放默认用户的信息等服务运行所必需的数据。因此数据库万一宕机或发生故障，就很可能会产生能够直接导致服务停止的严重故障，进而引起更严重的问题。

本节将介绍在数据库服务器停止工作的情况下，使数据库尽快恢复正常运行所采取的步骤。

导致数据库停止的原因

导致数据库停止的原因有很多，例如下面几项：

- 数据库服务器的进程（**mysqld**）异常结束
- 磁盘空间已满
- 磁盘故障
- 服务器电源故障

一般在 mysqld 异常结束的情况下，重启 mysqld 就能解决；在磁盘空间已满的情况下，将不使用的数据或文件删除以腾出空间即可。

另外在磁盘或电源等硬件发生故障的情况下，修复该故障往往要花一些时间。因为从替换故障硬件到配置服务器等修复工作往往要花费一些时间，若磁盘发生了故障，则其后的数据恢复工作也需要时间。

短时间内恢复的办法

以下考虑在硬件发生故障的情况下，短时间内恢复数据库服务器的方法。

假设有两台完全相同的数据库服务器，当一台因硬件故障无法使用时，可立刻切换到另一台。在此需要准备两台服务器，其中硬件及软件的结构和设置都相对比较简单，问题就剩下“数据库的数据”了。

这里登场的是同步（Replication）这种手段（图 2.3.1）。一般同步是指，将数据实时复制同步到其他位置的数据库。这里主要是通过 LAN 或 Internet 等网络，将存在物理性差异的服务器间的数据进行同步，即将不同服务器的数据库更新至一致。

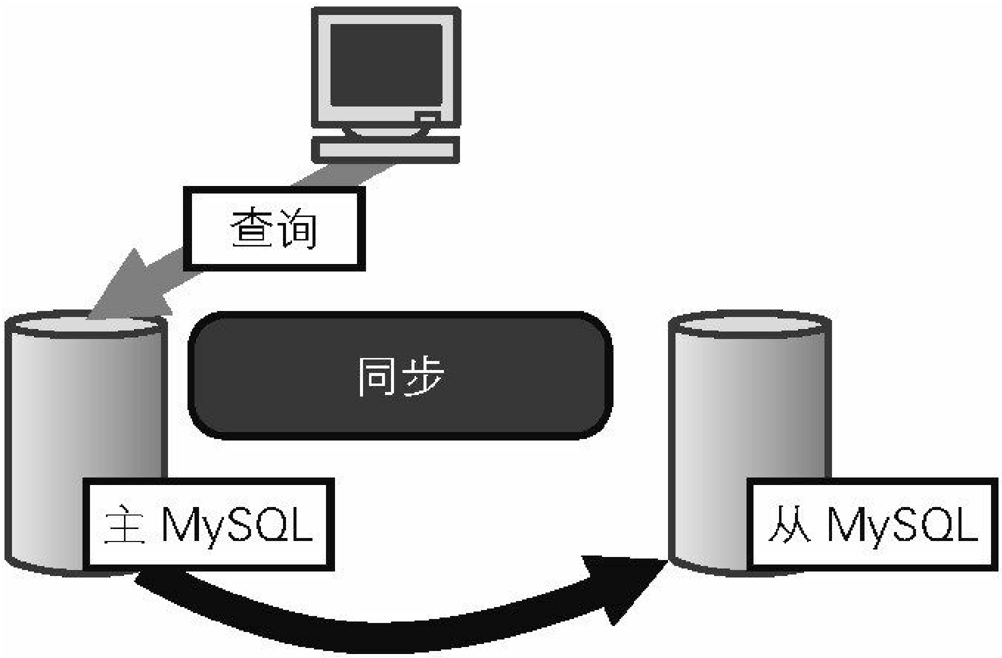


图 **2.3.1**　同步

总之，只要准备两台数据库服务器并同步其数据，当一台发生故障时就能实现短时间内恢复数据库服务[14]。以下将介绍 MySQL 的同步功能的特性与同步结构。

> [14] 为了尽可能地不让服务器发生灾难性后果，可以使用 RAID 提高单位服务器的可靠度，使其运作更加稳定。另外，使用配备了 Write Cache 的硬件 RAID，还可提高写入性能，可以说是一石二鸟。但请留意，根据产品的不同，有时不配备 BBU（Battery Backup Unit）的话就不能使用 Write Cache 的功能。

2.3.2　**MySQL** 的同步功能的特性和注意点

首先介绍 MySQL 同步的特性。本节所使用的 MySQL 版本为 5.0.45。

单主与多从

主（Master）指的是接收客户端发出的修改与查询两种类型的语句的服

务器；从（Slave）指的是不接收客户端发出的更新请求，仅通过与Master 的联动来进行数据的更新的服务器。

MySQL 的同步架构中，提供服务的有 1 台 Master 与多台 Slave（单主、多从）。

如果存在多个 Master，甚至需要在这些 Master 之间互相同步数据的话，这样的多主结构是不支持的 [15] 。另外因为可以存在多台 Slave，所以可以将包含有 SELECT 内容的查询语句分发到多台 Slave 进行处理以提高性能。详细介绍请参考 2.4 节。

> [15] 但是，通过分离更新表或记录的空间等，多主结构的应用也是没问题的。

异步数据同步

MySQL 支持“异步数据同步”。

所谓异步，是指系统更新到 Master 的内容并不一定会即时地反映到Slave（存在响应的时间间隔）。

虽说也存在支持复制同步的 RDBMS，但同步与异步各有优缺，并不能简单地说孰优孰劣。

被同步的数据内容

MySQL 是通过“SQL 语句”进行同步数据的。例如有一条 UPDATE 语句，那么无论该语句更新的内容是 1 项还是 100 万项，由 Master 传至Slave 的也仅有一条 UPDATE 语句。

该方式使 Master 与 Slave 之间只需交换较少的数据就能完成同步。但若在执行时同步了无效的查询语句（即不能返回正确结果的查询），则Master 与备份的 Slave 数据就可能会存在差异。

例如在修改类的语句中，若 LIMIT 语句没有使用 ORDER BY，Master数据库中 LIMIT 语句所选择的行与 Slave 数据库中所选择的可能会存在差异。因此，Master 数据库与 Slave 数据库中就会有不同的行被更新。该问题有一点非常致命，那就是不会察觉到数据已经出现了差异。如果幸运的话，在同步时会因违反 UNIQUE 等的规则而出现错误并停止，

这时就可能会觉察到；若是谁都没有察觉到异常，则很有可能存在数据不同步的危险。

SQL 的同步语句还存在一些其他的潜在问题，这些问题暂时都没有公认的解决办法，因此最好不要提交会出错的查询。

在 MySQL 5.1.5 以后的版本中，由于可以使用“以行为单位的同步功能”，所以该问题得以解决。在以行为单位的同步中，Master 数据库中被实际更新的行数据会被同步，因此无需使用前述的 LIMIT 语句，也避免了在执行时无从得知结果的问题。

在 MySQL 5.1.8 中还增加了“混合模式”。该模式一般以 SQL 语句为单位进行同步，根据情况的不同还可以以行为单位来执行同步操作。

2.3.3 同步的结构

接下来从以下几点来讲解同步的结构。

- **I/O** 线程与 **SQL** 线程
- 二进制日志与中继日志
- 位置信息

Slave 的 **I/O** 线程与 **SQL** 线程

为了实现同步，Slave 中设置了两个线程同时工作，即“I/O 线程”与“SQL 线程”。

I/O 线程是将从 Master 得到的数据（更新日志）放到被称为中继日志的文件中进行记录的。另一方面，SQL 线程则是将中继日志读取并执行查询。

为何要分为两个线程呢？这是为了降低同步的延迟。如果将 I/O 与 SQL 线程的工作只分配给一个线程，且 SQL 的处理又很花时间，那么在处理 SQL 语句期间是无法从 Master 复制数据的。为了避免这样的事态，应该将其工作分担给两个线程。

二进制日志与中继日志

Master 中会生成“二进制日志”（The Binary Log），Slave 中会生成“中继日志”（Relay Log）。

二进制日志中只记录修改数据的语句，而不记录查询类（不对数据库进行改动）的语句。另外，二进制日志除了被用于同步之外，还可被用于保存备份中更新的内容。但由于二进制日志不是文本格式，无法直接打开查看，需要在命令行下使用`mysqlbinlog`将其转换为文本格式。

中继日志是指，Slave 的 I/O 线程在从 Master 获取更新日志（记录了修改类语句所请求的数据）后，将其保存在 Slave 上。因此其内容与二进制日志相同。与二进制日志不同的是，当不需要时，中继日志会被 SQL 线程自动删除，不需要手动删除。

位置信息

因为 Slave 会记住复制同步了多少 Master 的数据，因此即便 Slave 的 mysqld 进程终止后隔段时间再启动，也可从终止时的断点开始继续同步数据。

在此我们将 Master 所在的主机名、日志文件名、日志文件中处理的信息称为“位置信息”。位置信息被保存在文本格式的文件“master.info”中，该文件可以使用 SQL 语句`SHOW SLAVE STATUS`进行确认。

2.3.4　搭建同步结构

接下来对同步结构的搭建流程逐步进行说明。

同步条件

使用 MySQL 搭建同步结构需要满足以下前提条件：

- **Master** 可拥有多个 **Slave**

  在一个 Master 下面能够配备多个 Slave

- **Slave** 只能挂靠一个 **Master**

  Slave 无法从多个 Master 中同步数据，但 Slave 在某些状况下也可以成为其他服务器（Slave）的 Master

- 所有的 **Master** 及 **Slave** 中必须指定不同的 **server-id**

  server-id 是同步结构中识别不同服务器的标志，需要指定为不同的数值

- **Master** 需要输出二进制日志

  由于要将修改类的请求传送到 Slave，因此需要保证开启 Master 的二进制日志

my.cnf

若需要搭建同步结构，可将 MySQL 的设定文件“my.cnf”参照代码清单 2.3.1 进行必要的设定。

代码清单 **2.3.1　my.cnf**

```
[mysqld]
server-id       = 1  ←❶
log-bin         = mysql-bin ←❷
log-bin-index   = mysql-bin ←❸
relay-log       = relay-bin ←❹
relay-log-index = relay-bin ←❺
log-slave-updates ←❻
```

需要将每个数据库服务器的 server-id 设置为不同的整数值，可指定的范围为 1 ～ 4294967295。由于代码清单 2.3.1 的 ❶ 中已经将 server-id 指定为了 1，若这是 Master 的 my.cnf，则 Slave 的 server-id 就需要指定为除 1 以外的其他值（例如 2）。

代码清单 2.3.1 的 ❷ log-bin 与 ❸ log-bin-index 是指开启二进制日志，并指定日志的文件名及索引的文件名。❹ relay-log 与 ❺ relay-log-index

也是同样，指的是开启中继日志并指定中继日志的文件名。

最后的 ❻ log-slave-updates 设定了 Slave 的二进制日志输出。若无该设定，则 Slave 不会输出二进制日志。但由于 MySQL 的同步操作中 Master 必须输出二进制日志，为了使 Slave 升格为 Master 的操作顺利进行，事先对 Slave 设定好二进制日志输出是较好的选择。

接下来将使用代码清单 2.3.1 的 my.cnf，启动搭建好的 Master 服务器的 mysqld。

建立同步专用的用户

为了连接 Master 与 Slave，需要在 Master 上建立用户，并赋予其最基本的 REPLICATION SLAVE 权限。该用户为同步专用，不需要给予该用户其他的权限。

例如可以将同步专用的用户名设置为“repl”，将该用户的密码设为“qa55wd”，若在 192.168.31.0/24 的网络上已经配置了 Slave，则可参考以下信息来配置 Master。在本命令中，为用户 rep1 赋予了 REPLICATION SLAVE 权限。

```
mysql> GRANT REPLICATION SLAVE ON *.* TO repl@'192.168.31.0/255.255.255.0'
```

同步开始时所必需的数据

在增加新的 Slave 时，或者在为遭遇故障的 Slave 引入备份设备时，Slave 的初始数据要怎么准备呢？这里不仅需要进行 Master 的完全转储，还需要明确位置信息，即需要知晓 Master 的二进制日志转储是何时进行的。因此，仅通过 `mysqldump` 工具获取数据的转储文件并不能搭建出 Slave。

简单起见，这里将转储 + 位置信息称为“快照”（Snapshot）。

在抓取快照时，若能停止 Master 的 mysqld，则先将 mysqld 停止，然后再使用 `tar` 等打包复制 [16] 包含有 MyISAM 和 InnoDB 数据文件的 MySQL 数据目录，或者使用 LVM（Logical Volume Manager）的快照

功能，减少花费的时间。使用 `tar` 的`--exclude` 选项可以排除不必要的文件（二进制文件等），从而缩短复制同步的时间。

> [16] 在 GNU tar 中，使用 -z 选项可同时进行 gzip 压缩。但根据数据容量的大小可能要花费一些时间，若希望减少时间，则建议不进行压缩而仅使用 tar 进行复制同步。

在此请注意不要忘记“记录位置信息”。

停止 mysqld 时，系统会记录下 Master 的二进制日志文件的文件名，并在启动后切换到下一个数字。即文件名为“mysql-bin.000002”的日志文件在启动后的位置信息将变为“mysql-bin.000003 的开头”[17] 。

> [17] 请注意二进制日志的头部的位置是“4”，而不是“0”。

在不能停止 mysqld 的情况下，只需先暂停使用修改类的语句（这时不让数据库修改数据），之后再使用完全转储，并记录好 `SHOW MASTER STATUS` 的结果便可以了。

另外，建议利用磁盘空间尽可能保存更多的快照。因为只要存在快照和抓取时间之后的 Master 的二进制日志，即便其再过时，也可以以此为基础制作 Slave，这对今后增设 Slave 及恢复故障设备都会起到很大的作用。

2.3.5　启动同步

下面基于快照制作 Slave。若 Slave 在 mysqld 正在运行时停止的话，可以采取前文中说明过的步骤将快照展开到 MySQL 的数据目录上。至此 Master 和 Slave 就存放好相同的数据了。

这里我们不立刻启动 Slave 的 mysqld，先来比较一下 Master 和 Slave 的 my.cnf 文件的差异。

Master 和 **Slave** 的 **my.cnf** 文件的差异

确认的重点是“server-id”，Master 和 Slave 之间只有 server-id 的值是必须不同的。若使用了 InnoDB，则需要将 Master 及 Slave 的 innodb_data_file_path 栏目中的数据文件的名称、数量、大小设置为同样的数值。

总而言之，只要确认 Master 和 Slave 的 my.cnf 中只有 server-id 的值不一样就可以了。

Slave 开始运行 **&** 确认

确认没问题的话就启动 Slave 的 mysqld，但因为仅仅启动的话还不能运行 Slave，所以需要在 Slave 中执行以下代码：

```
CHANGE MASTER TO  ← ❶指示与Master的关系
  MASTER_HOST     = 'my5-1',
  MASTER_USER     = 'repl',
  MASTER_PASSWORD = 'qa55wd',
  MASTER_LOG_FILE = 'mysql-bin.000003',
  MASTER_LOG_POS  = 4;

SLAVE START;  ← ❷开始复制
```

在❶ `CHANGE MASTER TO` 的“MASTER_LOG_FILE”与“MASTER_LOG_POS”中，指定快照抓取时的位置信息。

如果想要确认复制同步是否已经开始进行，可在 Slave 中执行 `SHOW SLAVE STATUS`，若结果中的“Slave_IO_Running”与“Slave_SQL_Running”都为 Yes，则表示没问题了。要是发生错误的话，可通过“Last_Error”或 MySQL 错误日志文件来调查产生错误的原因，将这些问题解决后，再执行`SLAVE START` 就行了。

2.3.6 确认同步的状态

最后来介绍确认同步的状态的方法。若同步不能正常运行，可通过监控同步的状态来查找具体原因。

Master 的状态确认

首先来介绍确认 Master 的状态的 SQL 语句。

- **——SHOW MASTER STATUS**

SHOW MASTER STATUS 指令可以被用来调查 Master 二进制日志的状态。执行结果如图 2.3.2 所示。项目内容见表 2.3.1。

```
mysql> SHOW MASTER STATUS\G
*************************** 1. row ***************************
            File: mysql-bin.000006
        Position: 98
    Binlog_Do_DB:
Binlog_Ignore_DB:
```

图 **2.3.2　SHOW MASTER STATUS** 的执行示例

表 **2.3.1　SHOW MASTER STATUS** 列出的各项目

项目名	内容
File	正在使用的二进制日志的文件名
Position	正在使用的二进制日志的位置信息
Binlog_Do_DB	设定记录二进制日志的数据库名称
Binlog_Ignore_DB	设定不记录二进制日志的数据库名称

- **——SHOW MASTER LOGS**

SHOW MASTER LOGS 会显示出目前 Master 中存在的所有二进制日志的文件名。执行结果如图 2.3.3 所示。

```
mysql> SHOW MASTER LOGS;
+------------------+-----------+
| Log_name         | File_size |
+------------------+-----------+
| mysql-bin.000001 |        117|
| mysql-bin.000002 |        463|
| mysql-bin.000003 |        343|
| mysql-bin.000004 |        242|
| mysql-bin.000005 |        117|
| mysql-bin.000006 |         98|
+------------------+-----------+
```

图 **2.3.3　SHOW MASTER LOGS** 的执行示例

由于二进制日志是渐渐增加的，需要定期清理。不过胡乱删除的话会导致同步中止，因此可以通过后面叙述的 `SHOW SLAVE STATUS` 对处理完毕的二进制日志文件名确认之后再删除。在有多个 Slave 运作的情况下，为了安全地删除，可以待所有 Slave 的二进制日志文件处理完毕后再开始进行。另外，也不能直接在文件系统上删除文件，而要在 Master 上执行`PURGE MASTERLOGS` 语句来进行删除。例如：

```
PURGE MASTER LOGS TO 'mysql-bin.000003';
```

执行以上语句会留下 mysql-bin.000003，而删除更早的 mysql-bin.000002 及 000001 日志文件。使用`RESET MASTER` 同样也可以进行删除操作，但执行该语句会删除 Master 所有的二进制日志，进而造成同步终止。在同步正在进行的情况下，不能使用`RESET MASTER`，而要 用`PURGE MASTER LOGS` 来删除日志。

Slave 的状态确认

以下介绍确认 Slave 状态的 SQL 语句。

- **——SHOW SLAVE STATUS**

执行`SHOW SLAVE STATUS` 就可以确认 Slave 的各种信息，如图 2.3.4 所示。各条目的含义如表 2.3.2 所示。`SHOW SLAVESTATUS` 输出的内容根据 MySQL 版本的不同可能存在差异，最新信息请参考 MySQL AB 的参考手册。

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
             Slave_IO_State: Waiting for master to send event
                Master_Host: my5-1
```

```
                  Master_User: repl
                  Master_Port: 3306
                Connect_Retry: 60
              Master_Log_File: mysql-bin.000006
          Read_Master_Log_Pos: 98
               Relay_Log_File: relay-bin.000116
                Relay_Log_Pos: 235
        Relay_Master_Log_File: mysql-bin.000006
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Master_Log_Pos: 98
              Relay_Log_Space: 235
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Master_SSL_Allowed: No
           Master_SSL_CA_File:
           Master_SSL_CA_Path:
              Master_SSL_Cert:
            Master_SSL_Cipher:
               Master_SSL_Key:
        Seconds_Behind_Master: 0
```

图 **2.3.4 SHOW SLAVE STATUS** 的执行示例

表 **2.3.2 SHOW SLAVE STATUS** 列出的条目（节选）

项目名	内容
Master_Host	Master的主机名
Master_User	连接Master所使用的用户名
Master_Port	Master的端口号
Connect_Retry	当连接Master失败的情况下，再尝试连接Slave的等待

	秒数
Master_Log_File	Slave的I/O线程正在读取的Master的二进制文件名
Read_Master_Log_Pos	在当前Master的二进制日志中，I/O线程已经读取的位置
Relay_Log_File	SQL线程当前正在读取和执行的中继日志文件的名称
Relay_Log_Pos	在当前Slave的中继日志中，SQL线程已读取和执行的位置
Relay_Master_Log_File	记录了SQL线程最后执行的查询的Master二进制日志的文件名
Slave_IO_Running	I/O线程是否被启动
Slave_SQL_Running	SQL线程是否被启动
Replicate_Do_DB	设定进行同步的数据库名称
Replicate_Ignore_DB	设定不进行同步的数据库名称
Last_Errno	最后执行的查询所返回的错误编号。“0”代表没有错误
Last_Error	最后执行的查询所返回的错误消息等。若Last_Error值是空值，则代表没有错误
Skip_Counter	最后使用SQL_SLAVE_SKIP_COUNTER时的值。若没有使用就设为“0”
Exec_Master_Log_Pos	SQL线程最后执行的查询在Master的二进制日志中的位置
Relay_Log_Space	所有原有的中继日志结合起来的总大小。单位是字节
Seconds_Behind_Master	测量SQL线程和I/O线程之间的时间差距，单位以秒计。若Master和Slave之间的网络连接较快，则能够十分近似地表示Slave比Master落后多少

具体输出的内容有很多，下面列举几个需要注意的地方。

首先来说日志文件名与位置信息，Master_Log_File 与 Read_Master_Log_Pos 相对应，Relay_Log_File 与 Relay_Log_Pos 相对应，Relay_Master_Log_File 与 Exec_Master_Log_Pos 相对应。

若 I/O 进程正常，Slave_IO_Running 就会显示为“Yes”；若 SQL 进程正常，Slave_SQL_Running 就会显示为“Yes”。若其中任何一方不是“Yes”，则同步就会终止，因此在监控 Slave 运行情况的时候需要密切关注上述这些条目。

Last_Error 中显示了错误信息，该错误信息会被记录在错误日志文

件中。正常情况下 Last_Errno 会显示为“0”；若出现错误，Last_Errnor 就会显示错误信息。在监控 Slave 的状态时，需要同时确认 Last_Errno 与 Last_Error 两者的情况。

2.4　MySQL 的 Slave+ 内部负载均衡器的灵活应用示例

2.4.1　MySQL 的 Slave 的运用方法

MySQL 的同步结构中的 Slave 对实时备份来说是一个非常有用的功能，但配置稍显复杂。本节中将会逐步探讨有关 MySQL Slave 的运用方法。

Slave 的运用策略

首先，实际操作中最常使用的是根据服务器属性的不同来规定其行为，以此来实现负载分发，即将修改类语句（INSERT、DELETE、UPDATE）在 Master 中进行，将查询类语句（SELECT）在 Slave 中进行。

为了进一步横向扩展，还可放置多台 Slave。MySQL 的同步虽然仅允许一台 Master 的存在，但设置多台 Slave 是完全可以的。在此建立多台 Slave，以将查询类语句分发到这些 Slave 上。

分发到多台 **Slave** 上

多台 Slave 的问题在于如何分发这些请求。在这里介绍两种方法以供各位参考。

- **——** ❶ 在应用程序端进行分发

  第一种方式就是在 Web 应用程序端进行分发处理。只要做出以下处理，即可实现应用程序端的分发。

  - 确定好所有 **Slave** 的主机名
  - 实现分发的逻辑，即如果决定分发到哪台 **Slave** 服务器上

- 实施 **Slave** 的状态监控，若宕机则不要分发到该 **Slave** 上

近期使用 O/R 映射访问数据库服务器的情况有很多，因此可以选择在 O/R 映射层配备此类分发处理。

- —— ❷ 在负载均衡器上进行分发

第二种方法是利用负载均衡器。提到负载均衡器，大家往往会想到是在外部客户端与 Web 服务器之间（即传送服务器文件的通道）所放置的设备，而既然负载均衡器是基于 Linux 部署的，所以不妨将其部署到服务器集群内部，这也就是第二种方案。

与在应用程序端进行分发相比，在负载均衡器上分发有以下优点：

- 应用程序端无需考虑 **Slave** 的台数

  由于增减 Slave 的台数可利用负载均衡器进行分发，因此就不需要特意使用 AP 服务器来分别管理所有的 Slave

- 应用程序端无需考虑 **Slave** 的工作状态

  由于负载均衡会进行恢复工作（通过监控分发集群的状态，万一发生情况就用备用的 Slave 顶替遭遇故障的设备），因此并不需要在应用程序端进行监控或分发处理

- 可以进行更加平衡的分发

  由于使用了“分发到连接数最小的 Slave 上”的策略，因此可实现更均衡地分配负载。在 Web 应用程序端进行分发时，若连接数超出了进程或 AP 服务器的负载，就很难实现均衡地分发

通过以该方式部署，不仅可以使应用程序端的处理减少，还可以使应用程序端不用得知 Slave 集群的状态。例如，在需要增加 Slave 的情况下，应用程序端不用做出任何准备，只要使用负载均衡器，通过下述的作业过程就能完成所需的处理。

因此，下文将对内部负载均衡器进行深入讲解。

2.4.2 通过负载均衡器将请求分发到多台 Slave 的方法

以下将介绍通过内部负载均衡器将请求分发到多台 Slave 的方法。

概况图

图 2.4.1 是相关的结构图。

- **AP**：发送请求的 **Web** 应用程序
- **db100**：**MySQL** 的 **Master** 数据库服务器
- **db101**、**db102**：**MySQL** 的 **Slave** 数据库服务器
- **db100-s**：**Slave** 集群所捆绑的虚拟 **Slave** 名
- **ll1**、**ll2**：内部负载均衡器。通过 **ll1**、**ll2** 构成 **VRRP** 的 **Active/Backup** 结构，**VIP** 为 **lls(192.168.31.230)**

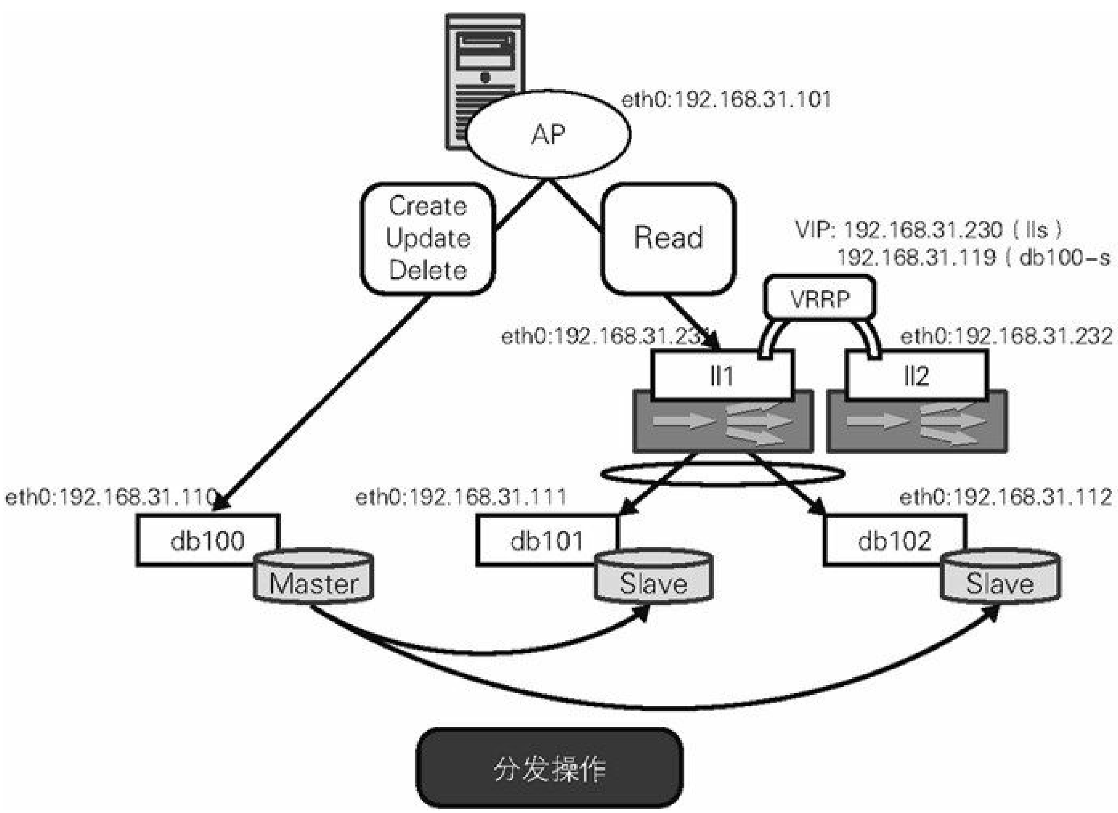


图 **2.4.1** 通过负载均衡器将请求分发到多台 **Slave**

在 MySQL 的分发架构中，存在一台 Master（db100）与两台 Slave（db101、db102）。处理客户端请求的 AP 将修改类的请求（Create、Updata、Delete）分配到 Master，将查询类的请求（不破坏数据的语句）分配到 Slave。而且并不直接连接到 Slave，而是经由内部负载均衡器（lls) 进行访问。内部负载均衡器 lls 能监控 Slave 集群是否正常工作，并恰当地将请求分发到 Slave 上。

本节使用的 MySQL 的版本是 5.0.45，keepalived 的版本是 1.1.15。

内部负载均衡器的配置

lls（ll1 与 ll2）的 keepalived.conf 的设定如代码清单 2.4.1 所示。

代码清单 **2.4.1　lls** 的 **keepalived.conf** 的设定

```
### basic section
vrrp_instance VI {
  state BACKUP
  interface eth0
  garp_master_delay 5
  virtual_router_id 230 ←❶
  priority 100
  nopreempt
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass himitsu
  }
  virtual_ipaddress { ←──┐
   192.168.31.230/24 dev eth0 │ ←❷
   192.168.31.119/24 dev eth0 │
  } ←────────────┘
}
### MySQL slave section
virtual_server_group MYSQL100 {
  192.168.31.119 3306
}
virtual_server group MYSQL100 {
  delay_loop   3
  lvs_sched    rr
  lvs_method   DR ←❸
  protocol     TCP

  real_server  192.168.31.111 3306 {
```

```
    weight 1
    inhibit_on_failure
    TCP_CHECK {
      connect_port 3306
      connect_timeout 3
    }
  }
  real_server  192.168.31.112 3306 {
    weight 1
    inhibit_on_failure
    TCP_CHECK {
      connect_port 3306
      connect_timeout 3
    }
  }
}
```

首先请看“basic”代码块。在此代码块中，设定了 lls 即负载均衡器的基本行为。需要留意的是代码清单 2.4.1 中 ❶ 处的 virtual_router_id 所指定的 VRID（Virtual Router ID，路由器集群识别标示符）。

在该 VRRP 中，虚拟路由器采用与 VRID 相同节点（路由）的集群结构（在 VRRR 协议中，VRID 是 VRRP 路由的唯一标识）。因此在同一网络的网段中，每个虚拟路由器集群的 VRID 都必须不同。若外部负载均衡器等已经存在虚拟路由器集群的话，请确保将 virtual_router_id 设定为不同的值。可以通过使用 tcpdump 查看 VRRP 报文，像图 2.4.2 那样查看实际正在使用的 VRID。

```
lls# tcpdump -n proto \\vrrp
00:59:42.164341 IP 192.168.31.231 > 224.0.0.18: VRRPv2, Advertisement, vrid
```

图 **2.4.2**　使用 **tcpdump** 查看 **VRRP** 报文

在 basic 代码块中还有一个重点，即代码清单 2.4.1 中 ❷ 处的 virtual_ipaddress。这里同时设定了内部负载均衡器自身的虚拟路由器地址（192.168.31.230）和虚拟 Slave（db100-s）所用的 IP 地址

（192.168.31.119）。

接下来是“MySQL slave”代码块，这里并没有什么需要特别注意的地方。virtual_server_group 中指定了虚拟 Slave 所使用的 IP 地址与端口号，下面的 virtual_server 中指定了真实服务器（Slave）。本例中是通过使用 TCP_CHECK 查看 TCP_CHECK 的 3306 端口是否启动来对真实服务器进行状态监控的，若需要更加严密的监控，可以写个脚本来确认实际发出的请求是否得到了回应，具体可以在脚本中通过 MISC_CHECK 等方式进行。

MySQL Slave 的设定

从内部负载均衡器的角度来看，必须对真实服务器，即 MySQL Slave 服务器进行配置。

虽然 MySQL 的服务并不需要进行特别的设置，但如代码清单 2.4.1 中的 ❸ 所示，因为设定了“DSR”的分发 [18]，所以必须确保虚拟 Slave 的 IP 地址能够正常收到报文。具体来说，需要对 Slave（db101、db102）执行以下命令：

> [18] 在 DRS 结构中，将 lvs_method 指定为“DR”，而不是“DSR”（请参考 1.3 节）。

```
iptables -t nat -A PREROUTING -d 192.168.31.119 -j REDIRECT
```

体验将请求分发到多台 **Slave** 的负载均衡

以上设定完毕后，即可体验负载均衡了。这里为了直观地进行确认，将分发对象 Slave 的 server_id[19] 结合其主机名 db101 与 db102 命名为 101 和 102。在访问该 server_id 时，将会把请求发送到虚拟 Slave（db100-s）上，下面我们来确认是否确实进行了分发处理（图 2.4.3）。

> [19] 通过 my.cnf 指定的 MySQL 的相关参数。在进行复制同步等操作的时候，通常被用于识别服务器。具体在 2.3 节中有说明。

```
w101$ check_lb_slave() {
> echo 'SHOW VARIABLES LIKE "server_id"' | mysql -s -hdb100-s
```

```
> }
w101$ check_lb_slave
server_id       101
w101$ check_lb_slave
server_id       102
w101$ check_lb_slave
server_id       101
```

图 **2.4.3**　体验将请求分发到多台 **Slave** 的负载均衡

虽然是向同一台虚拟服务器（db100-s）发送请求，但由于 server_id 的值不同，因此也可确认负载均衡是否完成了分发的工作。

2.4.3　内部负载均衡器的注意点……基于 DSR 的分发方法

相对于外部负载均衡器，内部负载均衡器也有需要特别注意的要点，即分发策略为 DSR（lvs_method DR），而不是 NAT（lvs_method NAT）。

在使用 NAT 策略的情况下，从客户端的角度来看，由于报文送出后会从不同的对象返回应答，因此无法再次处理返回的报文。

具体来说，假设有客户端向终点 VIP 地址发出请求报文，若使用 NAT 策略，在负载均衡收到该报文后，会把终点地址转换为真实服务器的 IP 地址再传送给真实服务器。虽然真实服务器在接到报文后会返回应答，但所返回的报文的起始地址却变成了真实服务器的地址。如果该返回的报文经由负载均衡器，则起始地址会转换为 VIP，不会引发问题；但若真实服务器与客户端存在于同一网络中，则没必要经过负载均衡器，而是直接将报文送到客户端上，这样就会导致送往 VIP 的报文由不同于 VIP 的地址（真实服务器的 IP 地址）返回了应答。

讲到这里想必大家都能领会了，没错，我们需要的正是一种名为 DSR 的报文的传输形式。只要使用 DSR 的分发方法，就不会有任何问题，都可以正常地响应该报文。虽然使用 NAT 的情况下稍加努力也能实现这种效果，但使用 DSR 可以明显减轻负载均衡器的负载，所以在使用内部负载均衡器时，没必要费尽心力地去应用 NAT 的拓扑结构。

2.5 选择轻量高速的存储服务器

2.5.1 存储服务器的必要性

在派发大容量文件（例如视频或音乐等）的服务中，内容文件如何存储时常是个很重要的课题。特别是在负载分发的环境下，同一个文件必须存放在多台 Web 服务器中，如果文件的数量及容量都非常大的话，则将面对以下问题：

- 部署到所有 **Web** 服务器将花费很多时间
- 必须在所有 **Web** 服务器上配置大容量硬盘
- 鉴于 **Web** 服务器上的所有文件都混杂在一起，根据需要将其分离存放比较麻烦
- 增设 **Web** 服务器比较麻烦（复制及同步文件很花时间）

虽然将所有的数据都存储在 MySQL 等的数据库服务器中会方便许多，但从取用与维护的便利性来考虑，在大多数情况下还是希望以文件的形式进行存储。在此情况下，通常的拓扑结构是使用大容量的存储服务器作为文件的存储，各 Web 服务器通过挂载这些存储服务器来将文件读出。

可惜的是，系统管理者通常会说“最好还是别用存储服务器吧”。其理由如下：

- 当存储服务器发生故障时通常殃及面很广
- 万一数据丢失，恢复要花费大量时间和精力
- 存储服务器易产生瓶颈
- 商用产品较为高价

存储服务器容易导致瓶颈，还容易造成单点故障。越是考虑发生问题时的棘手状况，对存储服务器的应用就越慎重。下文将探讨存储服务器发生故障时的细节。

存储服务器容易导致单点故障

在 Web 服务器挂载到存储服务器（NFS）上时，若存储服务器因故停止运作，将会造成很严重的事态。以下引用 `man nfs` 的介绍。

soft： 如果访问 NFS 文件的操作没有被服务器做出响应，将对调用的程序返回 I/O 错误。默认一直重试对文件的操作

hard：如果访问 NFS 文件的操作没有被服务器做出响应，将在控制台上显示“server not responding”，同时也会继续重试对文件的操作，直到服务器做出响应为止

intr： 如果访问 NFS 文件的操作超时，而且其 NFS 链接的命令为 hard，则允许中断 NFS 请求，在中断的情况下对应用程序返回 EINTR，默认不允许中断文件操作

……引自 `man nfs`

也就是说，在存储服务器停止期间，Web 服务器将会无限重试对 NFS 的文件操作。其结果会造成 Web 服务器访问文件的等待时间变长，进而造成无法浏览其他页面的情况（服务停止）。另外在文件操作不能中断（intr 没有被指定）的情况下，也会导致 Apache 无法重启。

虽说将挂载选项指定为 soft 或 intr 能够在一定程度上改善问题，但在 NFS 被作为操作系统的功能（文件系统）装载的情况下，调整 Web 应用程序的超时时间或中止文件操作并不可行。因此在文件操作超时前的等待时间中，若 Web 服务器的进程数超出限制，就会导致服务停止。

存储服务器容易造成瓶颈

存储服务器不仅容易导致单点故障，还易出现瓶颈的问题。虽然 Web 服务器的状态可以被监控，比如可以同时监控 10 台、20 台 Web 服务器的状态，但 NFS 服务器却不能被监控，所以此处若出现瓶颈，再去挽救是很困难的。如图 2.5.1 所示，虽然可以考虑增设 NFS 服务器的方法，但实际上不能解决的问题有很多。这是由于访问量集中的内容大多都是才更新的新内容，或者是具有某种推广效果的内容。也就是说，在访问比较集中的情况下，会有很多用户同时请求同一数据，即便根据目录对 NFS 服务器进行了区分，也依然会造成用户的访问请求集中于同

一台 NFS 服务器。

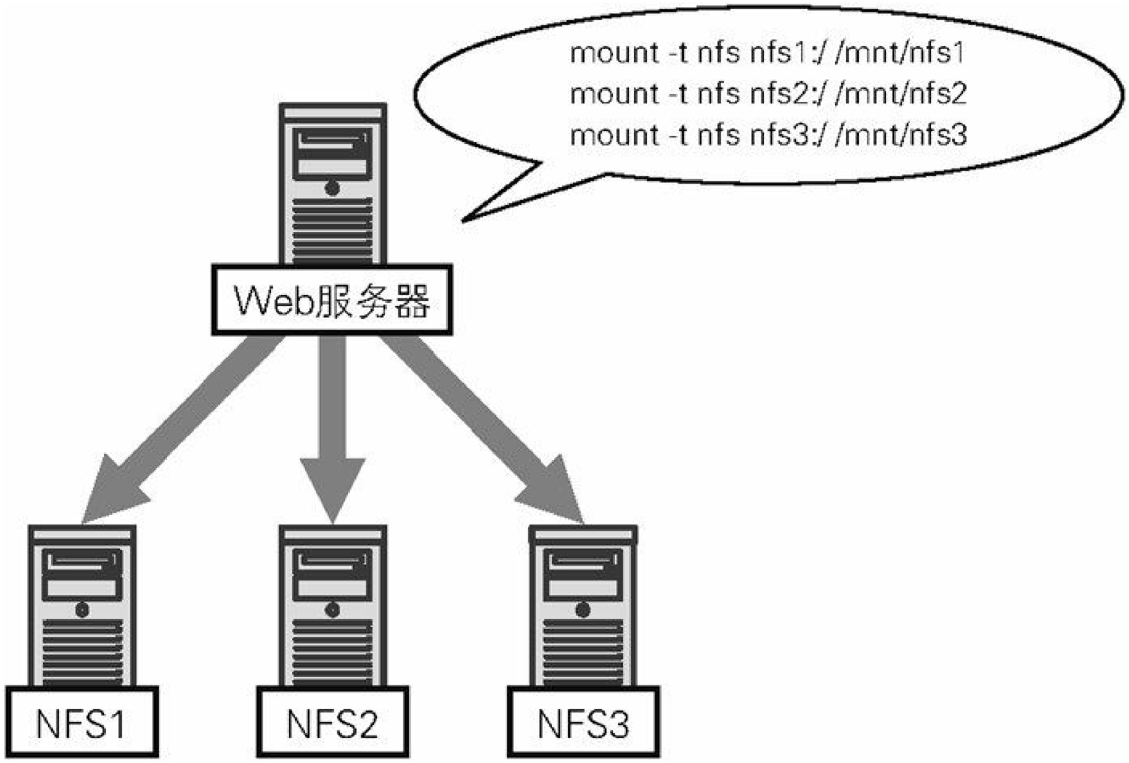


图 **2.5.1　NFS** 服务器的增设案例

另外，如果仅考虑负载问题的话，如图 2.5.2 所示，虽然可以根据 Web 服务器区分所挂载的 NFS 服务器，但这种拓扑结构会造成无法确保多台 NFS 服务器中文件的整合性。在打开内容时，必须向所有的 NFS 服务器传送同样的文件。而当文件数量达到相当庞大的规模时，确保所有服务器的内容相同将会是很困难的工作。

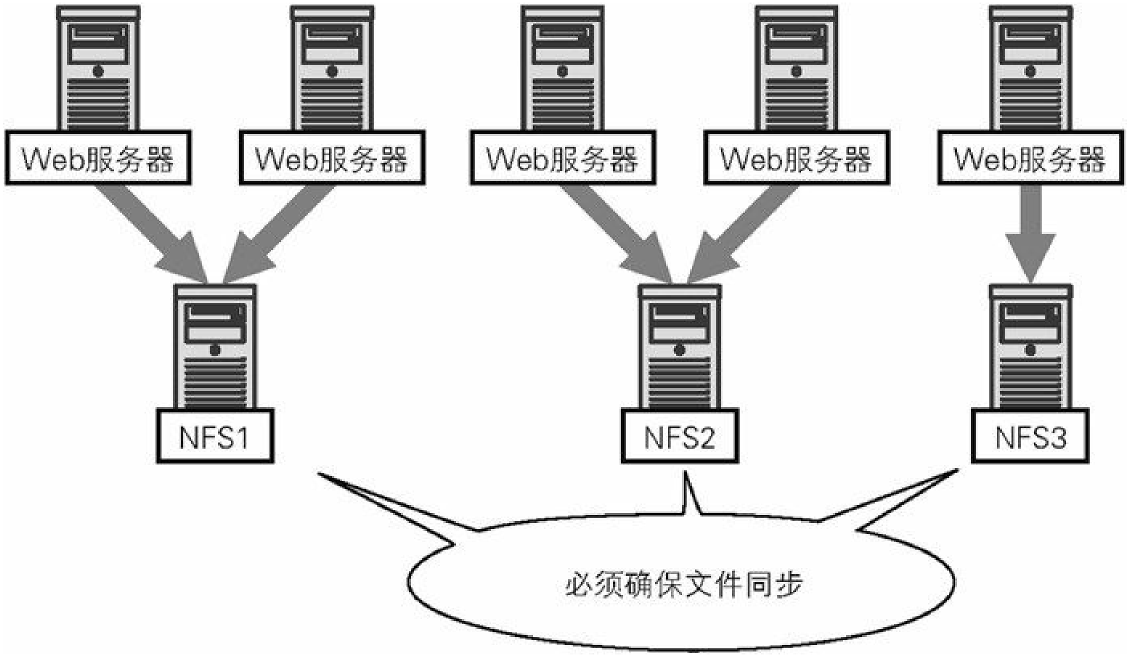


图 **2.5.2　NFS** 服务器的分发示例

2.5.2　理想的存储服务器

综上所述，理想的存储服务器究竟是怎样的呢？

- 在大量访问到来时也依然快速，不会出现瓶颈
- 能避免针对多台服务器的文件同步工作
- 能避免单点故障的出现
- 若能用开源软件实现就更好了

下文将逐步介绍满足以上要求的存储服务器 NFS 的建立。

减轻负载

就许多 Web 网站而言，最希望谋求的是存储服务器的“读取速度”和“磁盘容量”，而没必要追求存储服务器的“写入速度”。

需要高速写入的数据通常是“会话信息”以及“个人信息”等。由于会话信息是临时的数据，所以使用 memcached 等的缓存服务器即可；而个人

信息则放到数据库中更为合适。也就是说，只要存储服务器能大量存储视频、音频等较大容量的数据，且能够将必要的数据快速读出就可以了。

如此这般确实能大幅减轻 Web 服务器的负载量。在使用存储服务器时，例如在面向 NFS 的平台开发 Web 应用程序时，使用诸如 Jave 或 PHP 等语言，也可以像处理普通资料一样处理 Web 服务器上的资料，因此通过 HTTP 来操作文件这种方式，开发者想必也会很快适应的。

2.5.3　将 **HTTP** 作为存储协议使用

通过以上的说明可以看出，通过在存储服务器上搭建微型 Web 服务器，就可以大致解决性能方面的问题。我们在这里构建了如图 2.5.3 的系统，图 2.5.3 中的“WS”就是存储服务器。虽然文件的写入需要 NFS，但没必要在所有的服务器上都挂载，只需在将文件上传到服务器时挂载足以。图 2.5.3 的“Master”就是这样设置的，其他的服务器（Web 服务器）并不使用 NFS，而是在 WS 中经由 HTTP 来获取文件。

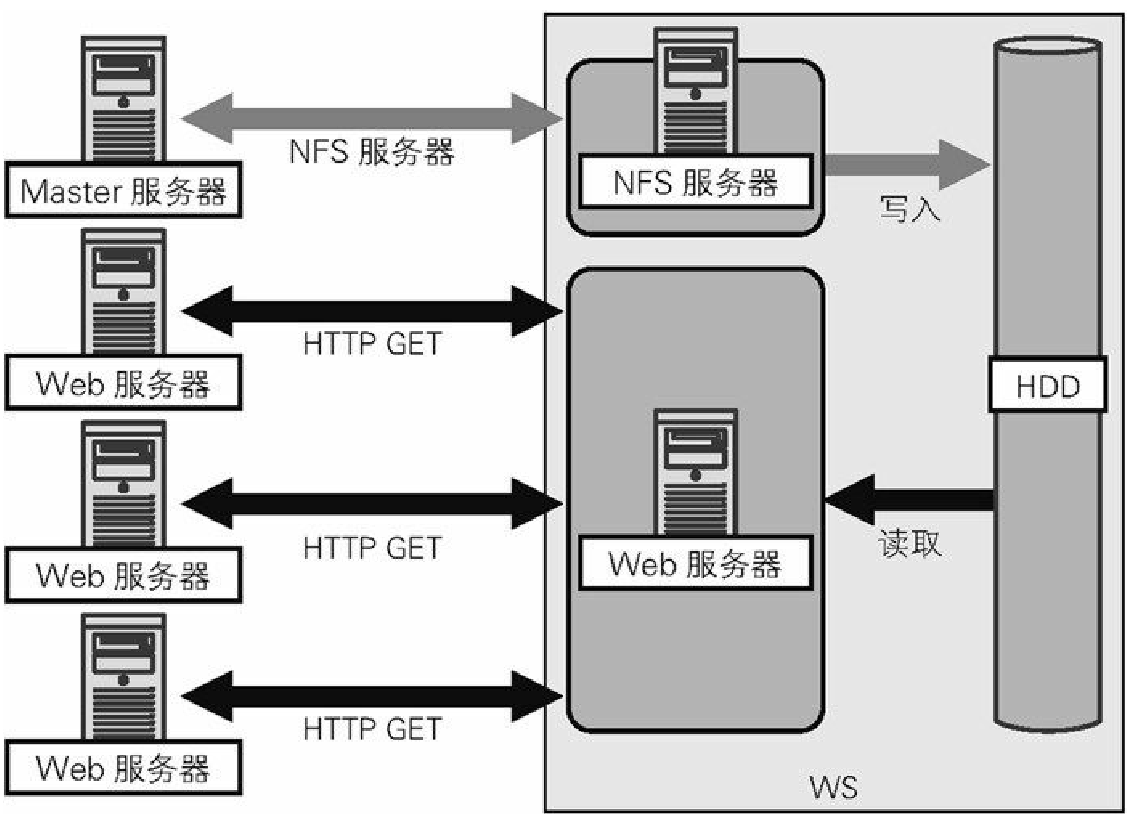


图 **2.5.3**　结合了 **NFS** 和 **HTTP** 的存储服务器

轻量 **Web** 服务器的选择

在 WS 中所使用的 Web 服务器均需要轻巧快速。即使没有 Apache 那样臃肿的各项功能也没有关系。因此需要舍弃 CGI、SSL 等动态网页生成的功能，只要确保能高速稳定地传输静态文件即可。在笔者的环境中，通过使用 **thttpd** [20] 来提供 HTTP 的访问，就能够显著提升性能。尤其是在并发量集中时段，更能表现其绝佳的处理性能。

[20] ***URL*** http://www.acme.com/software/thttpd/

在前文中提到了在访问量集中时，用户会集中请求同一数据，即会出现同一文件被反复读取的访问模式。该数据基本上都被缓存到了存储服务器的内存中。由于 thttpd 可以将存储服务器内存中的数据直接传送，因此并未加重存储服务器磁盘 I/O 的负担，这样便解决了存储服务器的瓶颈问题。

利用 **HTTP** 的优势

与 NFS 相比，可以说 HTTP 是服务器和客户端之间的桥梁。在 Web 服务器所挂载的 NFS 宕机的情况下，NFS 服务器停止会造成文件系统上的处理停止，即便重启 Apache 也无法解决该问题。

但若使用 HTTP 的话，在 Web 应用程序端就能自由地设定超时时间，从而很容易就能检查出存储服务器的异常情况并返回错误信息，以此便能在一定程度上回避因为存储服务器故障而引发整个网站停止服务的风险。

2.5.4　遗留的问题

至此我们已经实现了“即便出现大量访问也不会引起服务器瓶颈的高速存储服务器”，但仍不知如何“规避单点故障”及“避免多台服务器文件同步”。这两个问题是相互制约的，若不想造成单点故障就需要增设更多的服务器，但增设更多的服务器又会带来多台服务器文件同步的麻烦。要解决这些问题，需要更高级的策略。

这两个问题我们留到 3.2 节来解决。

专栏

选择小巧轻便的 **Web** 服务器

以下软件均为小巧轻便的 Web 服务器：

- khttpd（图 A） ***URL*** http://www.fenrus.demon.nl/

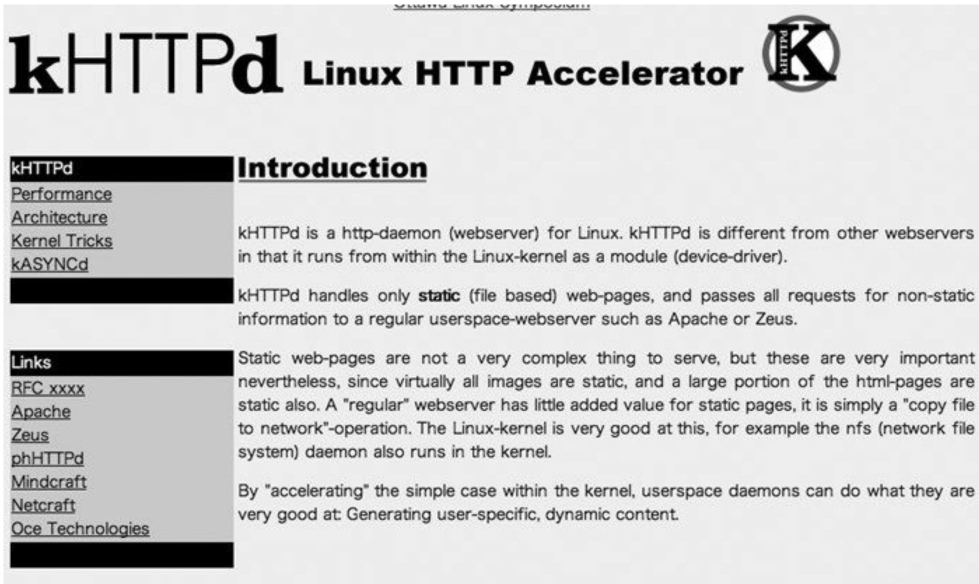


图 **A　khttpd**

- thttpd（图 B） ***URL*** http://www.acme.com/software/thttpd/

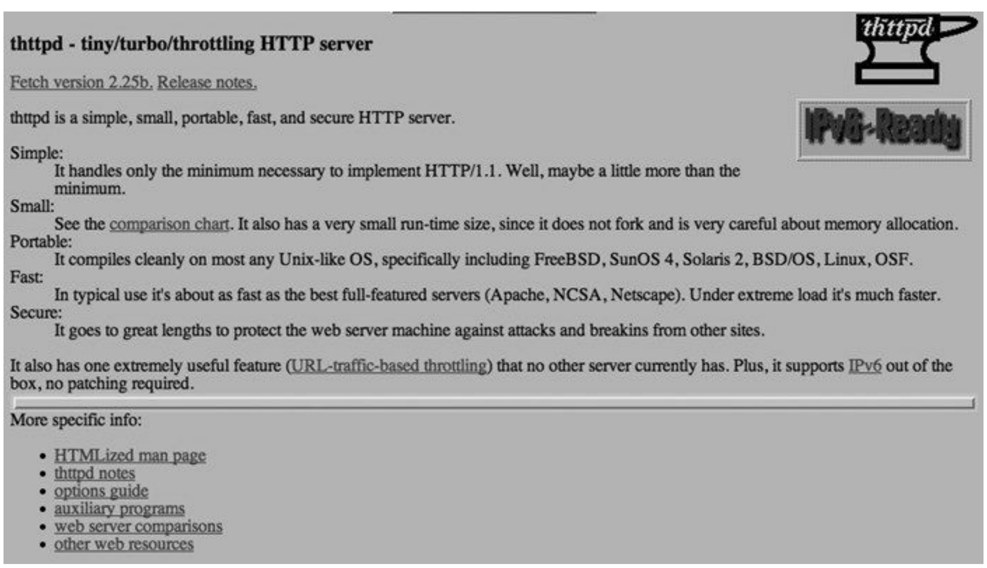


图 **B　thttpd**

- lighttpd（图 C） ***URL*** http://www.lighttpd.net/

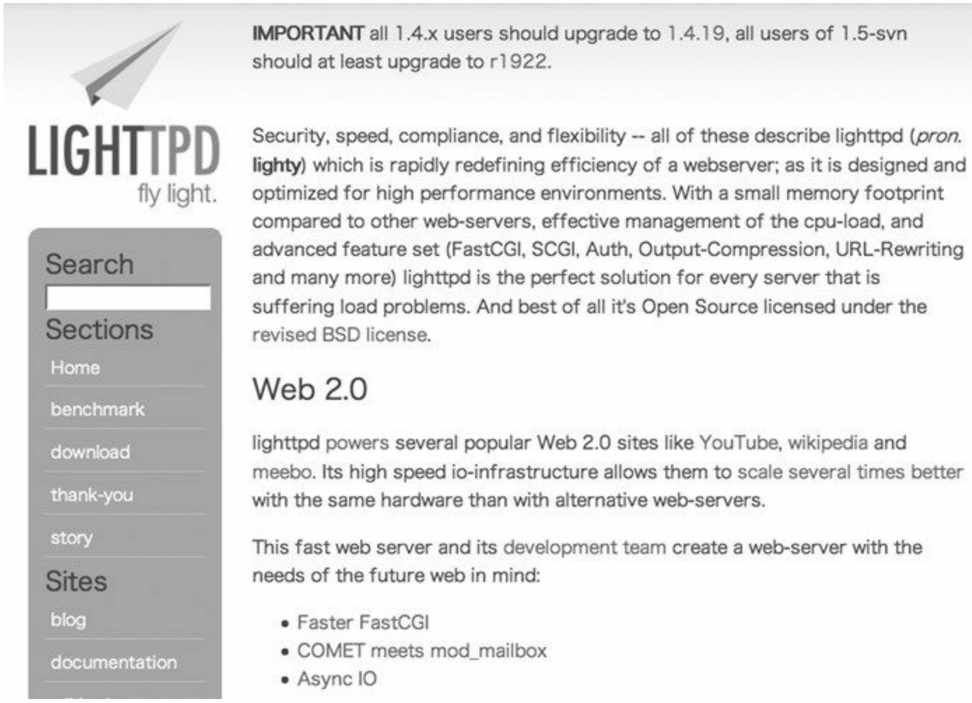


图 **C　lighttpd**

首先是 khttpd，它是作为 Linux 的内核模块安装的 Web 服务器。正是由于它是作为内核执行的，因此具备强劲的性能。但由于涉及内核可能会导致死机，本来可以期待在以后的版本中加以改善，但可惜的是在内核 2.5 的时候已经删除了相关代码，2.6 版时就已经完全消失了，因此笔者不得不放弃对它的使用。虽然此设想很值得玩味，但在内核空间处理应用程序协议方面好像还存在一些问题。

thttpd 和 lighttpd 都是轻巧快速的 Web 服务器软件。至于选择哪个确实很伤脑筋。根据笔者的亲自验证，二者在性能上几乎没有区别。虽然 lighttpd 的功能更多，但鉴于 thttpd 更简单易用，于是笔者选择了 thttpd。

第 **3** 章　进一步完善不间断的基础设施 **——DNS**服务器、存储服务器、网络

3.1　DNS 服务器的冗余

3.1.1　DNS 服务器冗余的重要性

虽说 DNS 服务器的故障并不那么常见，但一旦发生问题就要花很长时间才能查明原因。为了实现不间断的基础设施，将 DNS 服务器进行冗余处理十分必要。本节我们将围绕着下列几点来探讨 DNS 服务器的冗余。

- 利用解析库（**Resolver Library**）进行冗余，有降低性能的风险
- 基于服务器集群实现 **DNS** 的冗余

  → 利用 VRRP 的拓扑结构

  → DNS 服务器的负载分发

3.1.2　使用解析库实现冗余及存在的问题

为了实现 DNS 的冗余，可像图 3.1.1 那样在 /etc/resolv.conf 中指定多个 DNS 服务器。应用程序进行域名解析时使用的解析库，会以 /etc/resolv.conf 为准取得目的 DNS 服务器。具体在`man resolv.conf`中有下列说明，请留意有关同时指定多个 DNS 服务器的说明。

name server（域名服务器）的 **IP** 地址

name server 的网络上的 IP 地址（采用圆点记法），可以被解析。最多有MAXNS台（目前为3台，具体请查看<resolv.h>）可以被列出来，每个 name server 都对应有 name server 关键字，在列出多个 name servers

时，解析器将会按照所列出的顺序解析这些 name server（轮询）。当 name server 为空时，默认使用本地的 name server（这里使用的算法如下：首先尝试访问 name server，若访问超时，则尝试访问下一条 name server，直到最后一条。至此还未出现应答的情况下，将重复查询所有的 name server，直到达到最大重试次数。）

……引自man resolv.conf

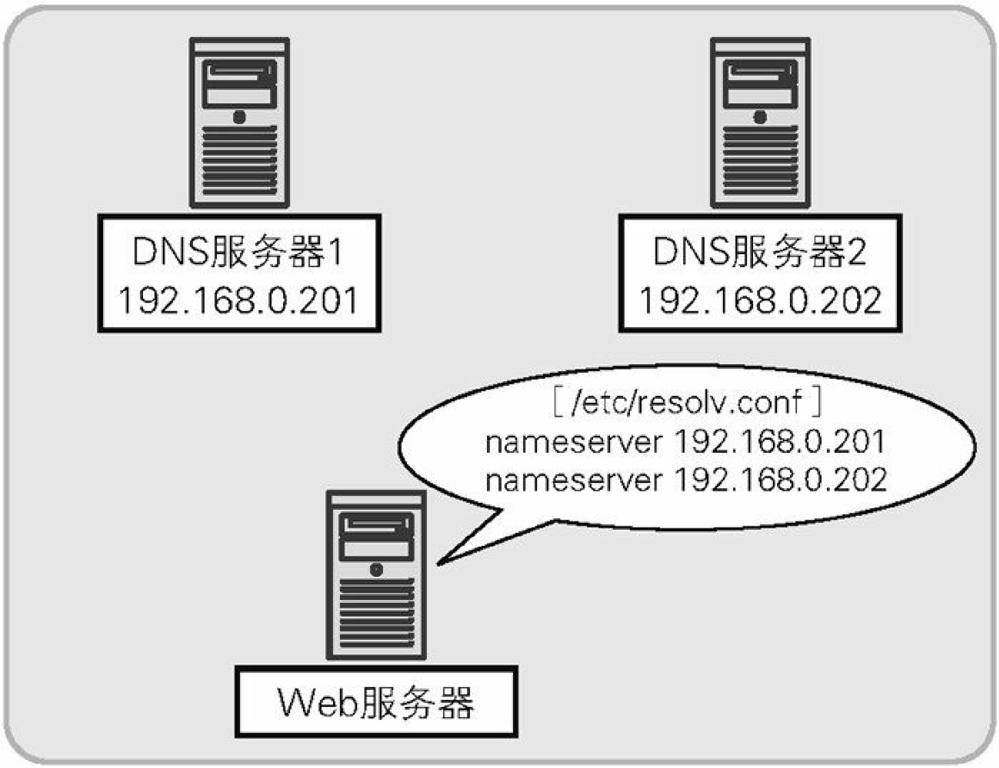


图 **3.1.1**　两台 **DNS** 服务器的拓扑结构

解析库存在的问题

因为事先指定了多台 DNS 服务器，若一台 DNS 服务器宕机，也不影响域名的解析。

但是，“若访问超时，则尝试解析下一条 name server”这一行为可能会带来一些问题。若首条指定的 DNS 服务器宕机，那必须要等超时过后（默认为 5 秒）才能轮到下一条服务器。这个等待的时间对于服务器集群来说是造成性能下降的原因，下面以一个简单的邮件服务器（Mail Server）为例来说明。

性能下降的危险性 **……**以邮件服务器为例

邮件服务器在发出邮件时，将访问两次 DNS，如下所示：

❶ 查询目的地址的域名所对应的 **MX** 记录

❷ 从 **MX** 的结果中查询 **A** 记录以获取目标服务器的 **IP** 地址

例如，假设此邮件服务器在 1 小时内必须发送 1000 封邮件，那么最低也要每 3 秒发送一封。若 ect/resolv.conf 指定的 DNS 服务器有一台停止工作，那么每次发送都将出现 5 秒钟的超时时间，即送出该信可能需要花费 10 秒钟，这样一个小时能够处理的邮件量也就 360 封而已，处理性能还不到期望值的一半。

这个例子虽然有点极端，但很能说明即使引入高性能服务器，也不能阻挡一台 DNS 发生故障时造成的性能下降。

DNS 故障会造成很大的影响

性能下降时不会出错，因此故障无法被及时发现。在上述邮件服务器的例子中，即使邮件服务器的性能显著下降，邮件发送工作也能完成，系统并未停止运作。因此当管理人员察觉到邮件服务器性能下降而去调查原因时，无论他如何检查邮件服务器的设定，可能也发现不了任何显性的错误。

DNS 服务器的故障不仅影响极大，而且故障原因的确定也很费时间，因此一定要多加注意。

3.1.3 基于服务器集群的 **DNS** 冗余

如前所述，解析库（DNS 客户端）在察觉到 DNS 服务器异常时，除了等待超时以外别无他法。因此我们应该避免将解析库冗余的技术应 用于服务器。

基于服务器集群的冗余将会实施 DNS 服务器端不下线的原则。下文将介绍使用 VRRP（请参考 1.4 节）的拓扑结构与使用负载均衡器的拓扑结构。

3.1.4　使用 **VRRP** 的拓扑结构

图 3.1.2 是使用 VRRP 实现 DNS 服务器冗余的拓扑结构。图中仅将 Web 服务器及邮件服务器在 /etc/resolv.conf 中设定为 VIP（192.168.0.200）。VIP 后端则是两台 DNS 服务器，其经由 VIP 实现了冗余。

图 3.1.2 的结构中利用了第 1 章介绍的 keepalived，各 DNS 服务器都预先安装了 keepalived。下面将以代码清单 3.1.1 为例来启动 keepalived.conf，首先启动的服务器将作为 Active 服务器被分配 VIP（192.168.0.200）。在两台服务器都启动的情况下，关闭 Active 服务器，即可确认是否能够正常启动故障转移。

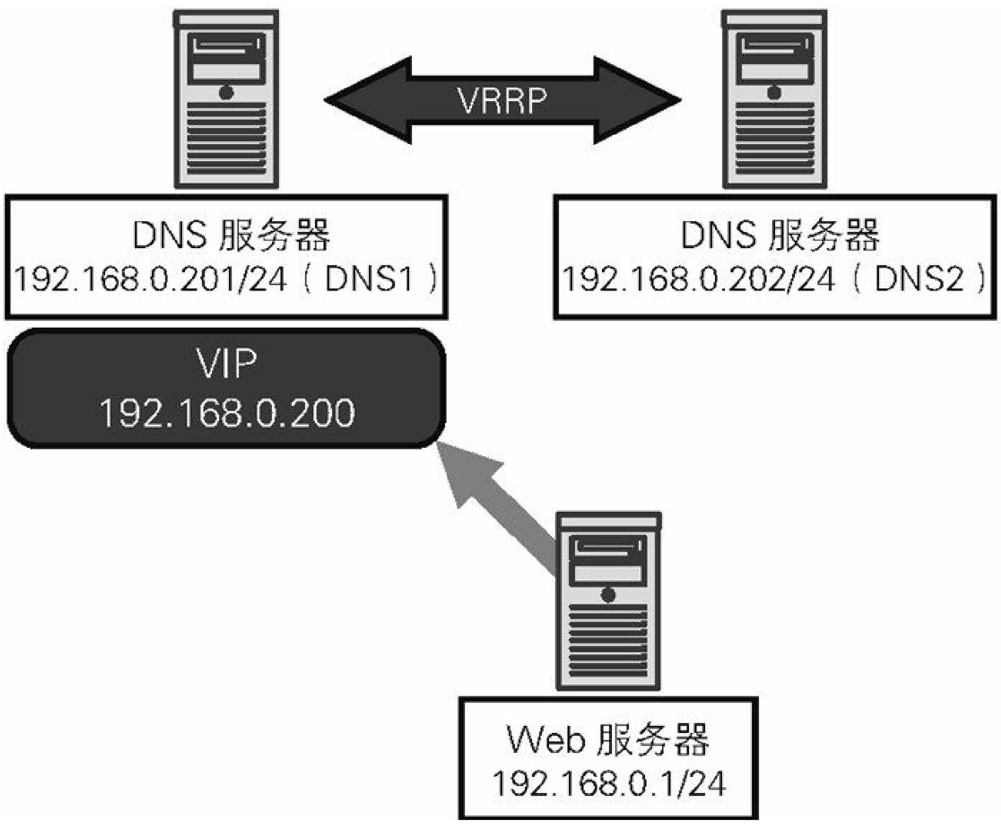


图 **3.1.2**　使用 **VRRP** 的冗余

代码清单 **3.1.1**　**DNS** 服务器的 **keepalived.conf**

```
vrrp_instance DNS {
  state BACKUP
```

```
  interface eth0
  garp_master_delay 5
  virtual_router_id 200
  priority 100
  nopreempt
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass HIMITSUDESU
  }
  virtual_ipaddress {
    192.168.0.200/24 dev eth0
  }
}
```

在该状态下，当 Active 服务器的 keepalived 停止时会发生故障转移，但并未发生 DNS 服务的故障转移。因此需要使用如代码清单 3.1.2 所示的健康检查脚本。在代码清单 3.1.2 的脚本中，每 5 秒都会使用 `dig` 命令确认一下自身的 DNS 情况，若`dig` 命令异常结束，则 keepalived 就会停止，这样当 DNS 服务器不可用时也能发生故障转移。

代码清单 **3.1.2　dns-check.sh**

```
#!/bin/sh
while true; do
    /usr/bin/dig +time=001 +tries=3 @127.0.0.1 localhost.localnet
    if [ "0" -ne "$?" ]; then
        /etc/init.d/keepalived stop
        exit
    fi
    sleep 5
done
```

在某些情况下，当主 DNS 服务器停止工作时，可能无法通过虚拟 IP 让备份服务器顶替主 DNS 服务器来完成工作（会提示超时），这种问题应该怎么解决呢？

鉴于 DNS 服务器的 BIND[1] 会在系统启动时通过分配到网卡 IP 地址来接收请求。因此，即便在 BIND 运行过程中被动态分配了 IP 地址，也无法响应发往该新的 IP 地址的 DNS 请求。所以在发生故障转移时，必须重启 BIND。

[1] BIND(Berkeley Internet Name Domain) 是类 UNIX 系统上实现的域名解析服务的软件包。

因为 Keepalived 在发生故障转移时可以运行任意命令，因此可以编辑 keepalived.conf 文件的 vrrp_instance 部分，添加下面的语句来解决这个问题。

```
notify_master "/etc/init.d/named restart"
```

3.1.5　DNS 服务器的负载分发

在 Active/Backup 的结构中，两台 DNS 服务器实际只有一台工作。为了充分利用服务器资源，接下来将介绍如图 3.1.3 所示的 **Active/Active** 的负载分发结构。

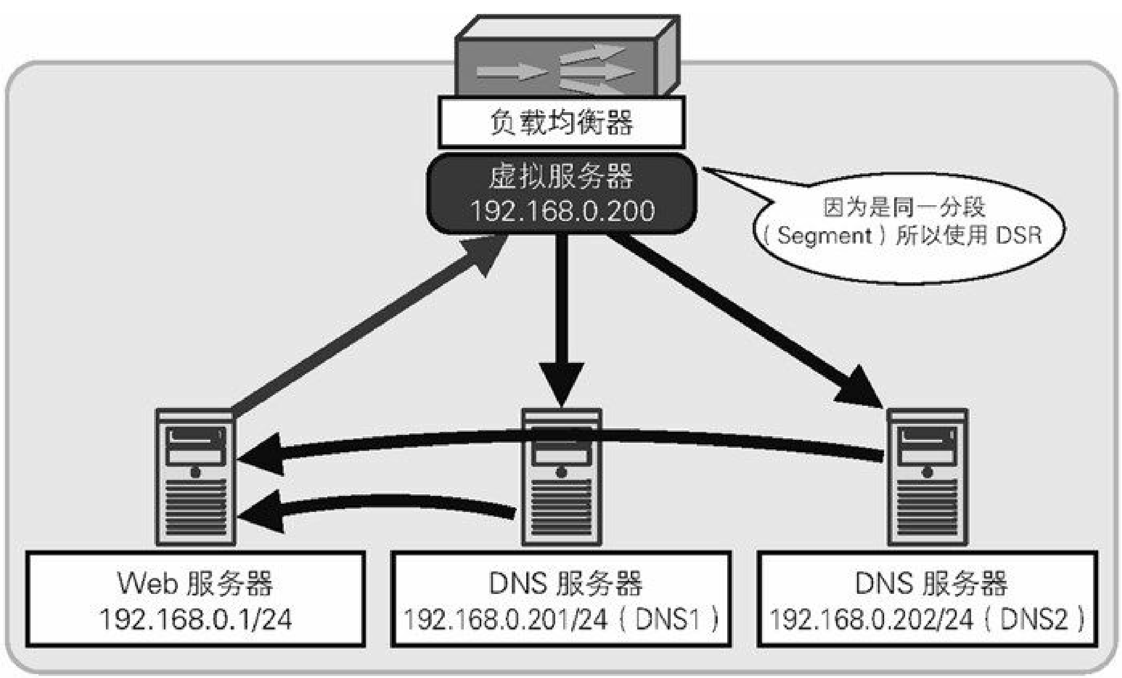


图 **3.1.3　DNS** 服务器负载分发的拓扑结构（**Active/Active** 拓扑结构）

与 Active/Backup 的拓扑结构不同的是这里利用了负载均衡器。Active/Backup 的结构中，每个 DNS 服务器都安装了 keepalived 并为其分配了 VIP；Active/Active 的拓扑结构中则将 VIP 分配给了负载均衡器。这无需更改 Web 服务器的设定，只需像之前那样预先将 VIP 指定到 ect/resolv.conf 中即可。由于同一子网上的负载分发不便选择 NAT 拓扑结构，因此应该使用 DSR 拓扑结构。另外，在各个 DNS 服务器中，为了能够处理发往 VIP 的数据帧，需要将 VIP（192.168.200/32）分配给回传接口（Loopback Interface），或者使用 iptables 进行重定向（Redirect）操作等。

这里在 Linux 中使用了 IPVS 及 keepalived 来搭建负载均衡器，具体请参考代码清单 3.1.3 中 keepalived.conf 的内容。由于 keepalived 并不支持 DNS 的健康检查，可以使用`dig` 命令通过确认 MISC_CHECK 状态来实现。

代码清单 **3.1.3**　负载均衡器的 **keepalived.conf**

```
virtual_server_group DNS {
  192.168.0.200 53
}
virtual_server group DNS {
  delay_loop 5
  lvs_sched  rr
  lvs_method DR
  protocol   UDP
  real_server 192.168.0.201 53 {
    weight 1
    MISC_CHECK {
      misc_path "/usr/bin/dig +time=001 +tries=3 @192.168.0.201
                 localhost.localnet"
      misc_timeout 5
    }
  }
  real_server 192.168.0.202 53 {
    weight 1
    MISC_CHECK {
      misc_path "/usr/bin/dig +time=001 +tries=3 @192.168.0.202
                 localhost.localnet"
      misc_timeout 5
    }
  }
```

```
}
```

3.1.6　小结

DNS 服务器在很多不显眼的地方进行着重要的工作。DNS 服务器所用的软件通常是比较稳定的，只有极少数会出现意外故障，因此人们对 DNS 服务器的故障应对可能并没有做过多考虑。

然而一旦 DNS 服务器发生故障，仅查明原因往往都会花费很多时间，为了不在故障发生时浪费太多精力，还是有必要提前做好准备的。

3.2　存储服务器的冗余 ———利用 **DRBD** 实现镜像

3.2.1　存储服务器的故障排解

存储服务器中存放了大量的文件，若硬盘故障造成数据丢失，恢复是很麻烦的。为了能有效恢复数据，及时备份是常规手段，但要恢复所有文件依然很花时间。而且一旦存储服务器发生故障，涉及范围通常会很广。所以一般会使用 RAID，这样即便硬盘发生故障，数据也不会丢失。

造成故障的原因也不仅仅是硬盘问题，RAID 控制器也有可能发生故障。当 RAID 控制器发生故障时，如果足够幸运，只需更换 RAID 控制器即可解决。若造成了无法写入硬盘等情况，则会有丢失数据的危险。

当然与磁盘相比，RAID 控制器发生故障的概率极低，但为了在发生故障时保护数据，也可以考虑准备两台存储服务器进行冗余处理。

3.2.2　存储服务器同步的难点

对存储服务器进行冗余的难点在于需要持续同步两台服务器的数据。作为最易实施的方法，可以采用“将数据上传两份分别到两个服务器上”的方式，但持续确保数据的完整性较为困难。如果上传程序不兼容，则有

可能导致只将文件传给了其中一台服务器，另外操作的失误也可能会造成只有一台服务器的数据进行了更新。

当文件数及容量较少的时候，尚可利用简单的脚本来机械地检查一致性，但当有成千上万个文件时，数据容量高达数百 GB（Gigabyte），这种情况下要做到此类检查就很困难。在无法检查一致性的情况下，若继续依赖此设备进行同步，可信赖性就会大大降低。

3.2.3 DRBD

在两台服务器上以文件为单位同步磁盘检查一致性时，文件数越多搜索目录就越花时间。在这种情况下，当磁盘超过负载时，整个服务器的性能就会大幅下跌，而使用 **DRBD** （Distributed Replicated Block Device，分布式复制块设备）[2] 技术即可解决该问题。

> [2] ***URL*** http://www.drbd.org/

以下是从 DRBD 官网引用的介绍。

> DRBD 技术为实现高可用的集群提供了专属的块设备。使用专用的网络，可以在两台电脑的块设备之间镜像数据。简单理解为网络上的 RAID1 结构即可。
>
> ……引自 http://www.drbd.jp/

DRBD 架构

如图 3.2.1 所示，DRBD 由 Master 服务器及备份服务器组成，分为以下两种拓扑结构：

- 内核模块（设备驱动程序）
- **Userland** 工具（控制程序）

DRBD 并非以文件单位来传送数据，而是针对块设备进行实时更新。因此在文件建立或更新时，无需理会 DRBD 及备份服务器的存在。

DRBD 的镜像是 Active/Backup 的拓扑结构。可以对 Active 端的块设备

进行数据的读写，而 Backup 端的块设备则不会被访问到。但在 8.0.0 版本以后，通过使用 OCFS（Oracle Cluster File System）及 GFS（Global File System）等的文件集群系统，开始支持 Active/Backup 的拓扑结构。虽然本书执笔时（2008 年 5 月）的最新版是 8.2.5，但笔者使用的依然是 0.7 版的环境，这是因为当时引入这项技术时最新版是 0.7。

以下介绍的拓扑结构以笔者系统中正在运行的 DRBD 0.7 版拓扑结构为准，关于8.2版中更新的功能及设定也会随时补充说明。另外，DRBD 0.7 版已经于 2008 年 10 月停止维护，在此之后若要引入这项技术，请使用最新版本。

3.2.4　DRBD 的设置与启动

代码清单 3.2.1 是运行 DRBD 所需的最低限度的设置。具体请在 /etc/drbd.conf 文件中建立 Master 服务器及备份服务器。

代码清单 **3.2.1　drbd.conf**

```
resource r0 {
  protocol A;
  on ws1 {
    device    /dev/drbd0;
    disk      /dev/sdb1;
    address   192.168.0.201:7789;
    meta-disk internal;
  }
  on ws2 {
    device    /dev/drbd0;
    disk      /dev/sdb1;
    address   192.168.0.202:7789;
    meta-disk internal;
  }
}
```

由于仅做了最低限度的设定，所以代码清单 3.2.1 显得很精简。具体各个项目的意义请参考表 3.2.1。

表 **3.2.1　drbd.conf** 的配置项目

项目名	内容
resource	定义资源的块，在此定义了名为“r0”的资源
protocol	指定数据传送的协议。可以指定为A、B、C三种，具体意义如下 • protocol A: 本地磁盘写入结束，数据被发往TCP缓冲时，表示写入操作完成（重视性能的异步传输） • protocol B: 本地磁盘写入结束，数据到达远程主机时，表示写入操作完成（介于A与C之间） • protocol C: 远程主机的磁盘上也完成写入操作时，表示写入操作完成（重视可靠性的同步传输）
on	定义各个主机的资源的块。这里指定的ws1与ws2是两个主机名。通过uname -n 的输出结果可判断目标主机（以on开头定义的主机名）是否取得了期望的配置结果
device	指定DRBD的逻辑块设备。在此指定的是运行mkfs和mount的块设备
disk	指定想要镜像的物理设备。可以指定任意块设备，若指定回环设备（Loopback Device ），则需要注意可能产生的问题
address	指定数据同步所等待的IP地址及其端口号。每个资源需要指定不同的端口号
meta-disk	指定存储元数据的设备。已经指定internal的情况下，需要指定disk的块设备为128MB以供元数据使用。在8.2版中，根据块设备的大小元数据的大小会发生改变

启动 **DRBD** 的 **Master** 服务器

接下来将启动 DRBD，首先在 Master 上进行以下操作，图 3.2.2 是实际操作的状态。

❶ 启动 **DRBD**

❷ 切换为 **Primary** 状态

❸ 在 **/dev/drbd0** 中建立文件系统

❹ 将 **/dev/drbd0** 挂载（**Mount**）到 **/mnt/drbd0**

```
ws1:~# /etc/init.d/drbd start
Starting DRBD resources:    [ d0 s0 n0 ].
..........
***************************************************************
 DRBD's startup script waits for the peer node(s) to appear.
 - In case this node was already a degraded cluster before the
   reboot the timeout is 0 seconds. [degr-wfc-timeout]
 - If the peer was available before the reboot the timeout will
   expire after 0 seconds. [wfc-timeout]
   (These values are for resource 'r0'; 0 sec -> wait forever)
 To abort waiting enter 'yes' [  5]:yes

ws1:~# drbdadm -- --do-what-I-say primary all  (0.7版
ws1:~# drbdadm -- -o primary all               (8.2版
ws1:~# mkfs /dev/drbd0
mke2fs 1.40-WIP (14-Nov-2006)
<中间省略>
This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
ws1:~# mount /dev/drbd0 /mnt/drbd0/
```

图 **3.2.2　DRBD** 的行为

启动 DRBD 后，首先请连接到目标镜像。在第一次安装时没有对象存在的情况下，会出现“`To abort waiting enter 'yes'`”这条信息并进入等待超时，这时输入“`yes`”以中断。启动完成后，DRBD 为“Secondary 状态”，此为等待数据从 Primary 状态的 DRBD 流入的待机状态，该状态下无法写入及读取块设备。

为了将文件系统进行挂载操作，需要使用 `drbdadm` 命令切换到 Primary 状态。通常切换到 Primary 状态需要使用 `drbdadm` 的 `primary` 命令，但在初次建立的情况下，还必须开启 0.7 版本的 `--do-what-I-say` 选项，或者 8.2 版本的 `-o` 选项。成功切换到 Primary 状态后，即可对 /dev/drbd0 和 /dev/drbd1 进行相同的操作了。

启动 **DRBD** 的备份服务器

同样在备份服务器上启动 DRBD，至此 Master 服务器与备份服务器的同步就开始了。具体情况可以参考图 3.2.3 进行确认。同步工作完成后安装也就结束了。如果在 Master 服务器的 /mnt/drbd0 中建立文件并写入数据，数据就会被传送到备份服务器。

```
ws2:~# /etc/init.d/drbd start
Starting DRBD resources:    [ d0 s0 n0 ].
ws2:~# cat /proc/drbdop
version: 0.7.25 (api:79/proto:74)
GIT-hash: 3a9c7c136a9af8df921b3628129dafbe212ace9f build by root@
ws2, 2007-12-31 22:20:38
 0: cs:SyncTarget st:Secondary/Secondary ld:Inconsistent
    ns:0 nr:528 dw:528 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0
        [>...................] sync'ed:  0.7% (666832/667360)K
        finish: 0:27:47 speed: 264 (264) K/sec
```

图 **3.2.3**　确认 **DRBD** 的同步

3.2.5　DRBD 的故障转移

DRBD 在 Master 服务器发生错误时，并不会自动切换到备份服务器，因此需要使用 keepalived 来实现故障转移。

手动切换

在实现自动进行故障转移前，首先请尝试手动切换。为了实现故障转移，需要将 Master 的 DRBD 切换为 Secondary 状态。为避免块设备挂载失败，需要事先停止 NFS 服务器（即网络文件服务器）并取消挂载。本处理的脚本如代码清单 3.2.2 所示，该脚本应该保存在两台服务器的 /usr/local/sbin/drbd-backup 中。

代码清单 **3.2.2**　**drbd-backup**

```
#!/bin/sh
/etc/init.d/nfs-kernel-server stop
umount /mnt/drbd0
```

```
drbdadm secondary all
```

将备份服务器作为 Master 服务器使用时，需要将 DRBD 切换到 Primary 状态，并在确认挂载块设备后启动 NFS 服务器。本脚本如代码清单 3.2.3 所示，保存在两台服务器的 /usr/local/sbin/drbd-master 中。

代码清单 **3.2.3　drbd-master**

```
#!/bin/sh
drbdadm primary all
mount /dev/drbd0 /mnt/drbd0
/etc/init.d/nfs-kernel-server start
```

为了更加容易地确认数据同步，需要事先在 Master 的 /mnt/drbd0 中创建恰当的文件。然后在 Master 服务器上执行`drbd-backup` 命令，将两台服务器切换为 Secondary 状态。最后在备份服务器上执行 `drbd-master` 命令，在切换到 Primary 状态后，/dev/drbd0 就会被挂载到 /mnt/drbd0 上。

至此应该已经在 Master 服务器上建立了适当的文件。若以后 Master 服务器发生故障，将自动执行这些步骤触发故障转移。下文将介绍利用 keepalived 的 VRRP 功能来实现 NFS 服务器冗余的方法。

keepalived 的配置

图 3.2.4 是将 NFS 服务器通过 VRRP 实现冗余的例子。代码清单 3.2.4 是本拓扑结构中 keepalived 的配置。VIP 为 192.168.0.200，NFS 客户端 192.168.0.200:/mnt/drbd0/ 已经挂载在了 NFS 上。即使服务器发生故障转移，NFS 客户端也无需重新挂载。

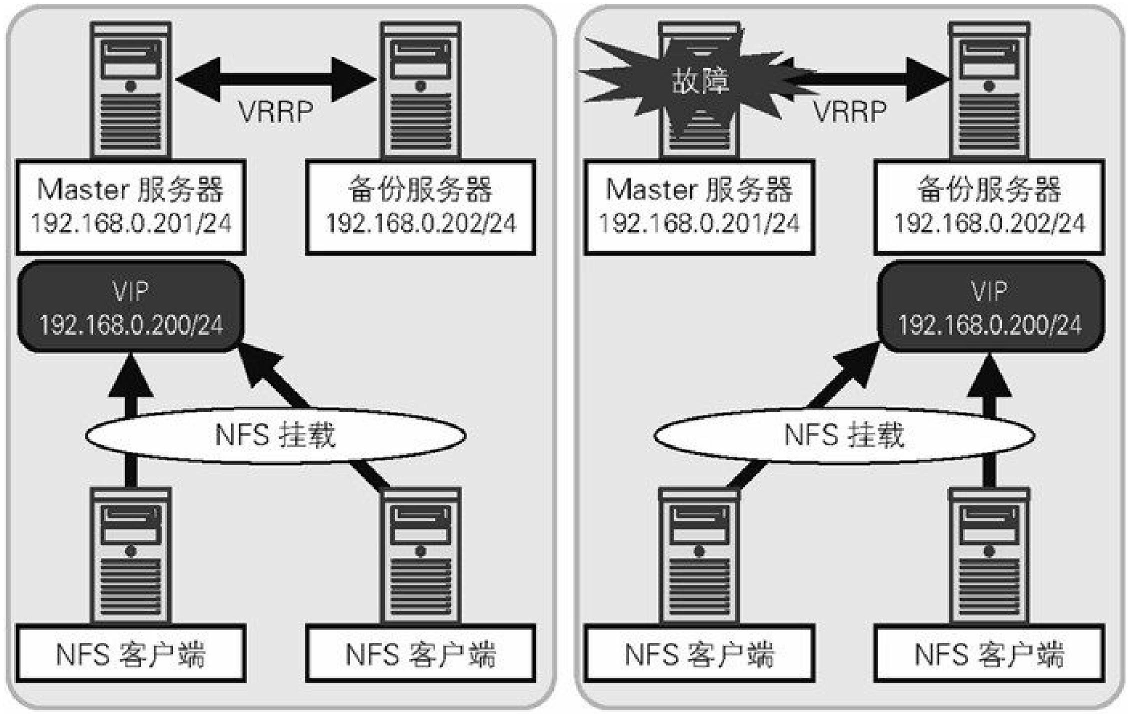


图 **3.2.4　NFS** 服务器的冗余

代码清单 **3.2.4　keepalived.conf**（**DRBD** 用）

```
vrrp_instance DRBD {
  state BACKUP
  interface eth0
  garp_master_delay 5
  virtual_router_id 200
  priority 100
  nopreempt
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass HIMITSU
  }
  virtual_ipaddress {
    192.168.0.200/24 dev eth0
  }
  notify_master "/usr/local/sbin/drbd-master"
  notify_backup "/usr/local/sbin/drbd-backup"
  notify_fault  "/usr/local/sbin/drbd-backup"
}
```

在此初次出现的参数如表 3.2.2 所示。

表 **3.2.2　keepalived.conf** 的配置项目（新出现的）

项目名	内容
nopreempt	将 VRRP 的抢占模式设为无效。关于抢占模式的详细介绍请参考第 1 章（其目标是避免任何不必要的故障切换）
notify_master	当 VRRP 为 Master 状态时，指定想要执行的命令
notify_backup	当 VRRP 为备份状态时，指定想要执行的命令
notify_fault	当网络接口的链接断开时，指定想要执行的命令

notify_master 及 notify_backup 已经在之前的代码清单 3.2.2、代码清单 3.2.3 中进行了指定，因此在故障转移时可以更改 DRBD 的状态。在本设定中，当备份服务器切换为 Master 服务器时，drbd-master 会被执行并触发故障转移。

但万一 keepalived 停止运作，将会导致 notify_master 及 notify_backup 的脚本无法运行。若 Master 的 keepalived 停止运作，Master 的 DRBD 将在 Primary 的状态下发生故障转移，进而就会导致备份服务器的 drbd-master 出现错误。因此，在 keepalived 停止运作时，一定要确保执行 drbd-backup 的脚本。

利用 **daemontools** 来控制 **keepalived**

为了解决该问题，使用代码清单 3.2.5 作为 keepalived 的启动脚本。但此处并非直接启动 /etc/init.d/keepalived，而是使用 daemontools 的 run 脚本。具体将在 5.4 节中讲解，daemontools 中通过这样的脚本可以控制守护程序的启动。在本节中，我们将使用 `wait` 等待 keepalived 结束，`wait` 完毕后就会执行 drbd-backup 脚本。由于从 supervise 发来 的

信号会经由 `trap` 命令传送到 keepalived 中，因此同样也可利用 daemontools 来直接控制 keepalived 的操作。通过使用此工具，即便 keepalived 由于什么原因停止运作了，也一定可以执行 drbd-backup 脚本。

代码清单 **3.2.5　keepalived** 的启动脚本

```
#!/bin/sh
[ -f /var/run/vrrp.pid ] && exit
exec 2>&1
trap 'kill -TERM $PID' TERM
trap 'kill -HUP  $PID' HUP
trap 'kill -INT  $PID' INT
/usr/local/sbin/keepalived -n -S 1 --vrrp &
PID=$!
wait $PID
/usr/local/sbin/drbd-backup
```

3.2.6　NFS 服务器故障转移时的注意事项

由于 DRBD 所镜像的设备是 NFS 共有的，因此不能在 Master 服务器上启动 NFS 服务器。实施 NFS 服务器的冗余时有可能出现不同于 Web 服务器及邮件服务器冗余时所遭遇的问题，因此需要多加留意。因故障转移而成为新 Master 的 NFS 服务器并没有被任何客户端所挂载，若此时没有向 NFS 客户端声明故障转移导致了服务器切换，NFS 客户端就会认为已经进行了挂载并访问文件，但这时 NFS 服务器则会认为没有挂载的客户端发起了文件请求，从而拒绝该请求。为了解决该问题，可使用以下方法。

- 同步 **/var/lib/nfs/**

  NFS 服务器的连接信息被存储在 /var/lib/nfs/ 下，DRBD 将该参数进行镜像操作，以便在故障转移时能实现正常切换。但根据分发策略（Distribution），NFS 服务器的启动脚本中有时是使用`exportfs` 命令来清除连接信息的，因此需要额外创建不清除连接信息的脚本，并在发生故障转移时通过该脚本启动 NFS 服务器

- 使用 **nfsd** 文件系统

  nfsd 文件系统 Linux 的固有功能，专门被用来支持 NFS 服务器的冗余。在执行`mount -t nfsd nfsd /proc/fs/nfsd`命令的状态下启动的 NFS 服务器并没有使用 /var/lib/nfs/ 目录，因此即便是完全陌生的 NFS 客户端发来访问请求，也会像该客户端已经挂载了那样来进行处理。在 Linux 2.6 版内核中使用该文件系统来进行操作相当便利

3.2.7 备份的必要性

即便在 DRBD 中将磁盘做了镜像处理，也依然不能保证 100% 的安全，例如将文件误删后依旧难以找回。镜像是 DRBD 的优点，但万一文件被误删，就立刻会被反映到备份服务器上去，从这一层面上来看，镜像也是一个致命的弱点。因此虽然很花时间，但还是做好最坏的打算，务必做好数据的备份，不过也不用每天都进行备份。

3.3 网络的冗余 **——**驱动绑定、**RSTP**

3.3.1 **L1/L2** 上部件的冗余

前两章以及第3章的前几节都在探讨 OSI 参考模型下的第三层（L3=IP 层）与第七层（L7= 应用层）的冗余问题。但若下层的物理网络（L1= 物理层）或以太网（Ethernet）级别的通信发生故障，即便上层正常运作，也会导致整个系统的故障。

本节中将讲解在 L1 及 L2 的结构元素发生故障时系统也不会停止的冗余策略。通过对 L1/L2 做出冗余处理，不仅能避免故障，还能细化系统维护层面的管理 [3] 。

> [3] 例如在笔者管理的环境中，就曾对所有的交换机及负载均衡器兼路由器都做过不间断的处理。

3.3.2 故障点

在 L1/L2 的结构元素中，通常是以下原因导致故障的：

❶ **LAN** 网线

❷ **NIC**（网卡）

❸ 网络交换机的端口

❹ 网络交换机

LAN 网线故障是因为断线或线缆接触不良造成的，以笔者的经验来看，通常表现为网络连接的上行及下行都出现问题。与网卡连接的网络交换机的特定端口也有可能发生故障。当然网络交换机自身也有可能发生故障。例如不小心碰掉了网络交换机的电源，虽然这种故障有点低级……

接下来将仔细探讨各故障元素。❶ ～ ❸ 是服务器与网络交换机之间的连接方面的故障，在此统称为“连接故障”。

而交换机与交换机之间也会发生 ❶ 和 ❸ 那样的故障，在此统称为“交换机间的连接故障”。

像 ❹ 那样的网络交换机故障则被称为“交换机故障”。

下文将会就如何避免这些故障进行一系列探讨。

3.3.3　链路冗余与驱动绑定

为避免连接故障，需要将服务器与交换机之间的连接进行冗余处理。也就是说，需要在服务器上准备多块 NIC（网卡），同时将 LAN 网线连接到这些网卡上。当然也不仅仅只是准备多块网卡而已，为了能使用这些网卡进行通信，还需对每个网卡分配 IP 地址。这样一来，每当连接到网络的主机之间进行通信时，都需要诊断哪个网卡可用，据此来切换目的地址，但这几乎是无法做到的。为了解决该问题，Linux 中提供了驱动绑定 （Driver Binding）这样的工具。

驱动绑定

驱动绑定 [4] 是 Linux 自带的网络驱动的组件之一。驱动绑定将多块物理网卡（物理 NIC）进行托管，以作为单块逻辑网卡（逻辑 NIC）使用。

> [4] 关于驱动绑定，Linux 内核的附属文档是最主要的参考信息，请在 kernel.org 下载 所发布的软件包，并访问其中的 Linux-2.6.X.X/Documentation/networking/bonding.text。

逻辑网卡不但可以作为赋予地址的对象网卡来使用，还可以作为 Linux 内核的 IP 别名（IP Alias）功能、VLAN（Virtual LAN）功能、桥接功能等的对象网卡来使用。在使用逻辑网卡进行通信时，根据驱动绑定中的设置会将请求分配到物理网卡上，并检查分配的目标物理网卡是否存在故障，以避免将请求分配到有故障的物理网卡上。

驱动绑定从多个物理网卡中选择一个用于通信时，可使用多种模式，具体可参考表 3.3.1 的选择项 [5] 。但无论选择哪个模式，若物理网卡发生故障，都没法让该网卡恢复使用。

> [5] 在这些模式中，active-backup 是使用方法最简单的模式，笔者管理的环境中使用的就是 active-backup 模式。

表 **3.3.1** 驱动绑定的行为模式 ※

模式	行为
balance-rr	根据需要送达的数据帧切换物理网卡（轮询）
active-backup	若该物理网卡可用，则只使用该网卡。若该网卡发生故障，则切换到下一块网卡
balance-xor	将发出请求与接收请求的 MAC 地址进行 XOR（异或）运算后再决定使用哪个物理网卡
broadcast	将需要发送的数据帧复制后，发送到所有物理网卡上，这些数据帧均相同
802.3ad	使用 IEEE 802.3ad 协议，在交换机之间动态创建链路聚合（Link Aggregation）
balance-tlb	选择负荷量最低的物理网卡进行发送。接收请求使用特定的物理网卡

balance-alb	选择负荷量最低的物理网卡进行发送及接收

※ 引自 linux 2.6.24 版内核的 bonding.txt 文件。

驱动绑定还有个很重要的参数，可以监控物理网卡的故障。有 **MII**（Media Independent Interface）监控 和 **ARP** （Address Resolution Protocol）监控 两种监控种类可供选择。

MII 监控是监控可能出现的物理网卡链路故障（Link Down），虽然可以很高效地在短时间内检查出故障，但却无法监控出网卡链路正常（Link Up）时出现的通信故障 [6] 。

> [6] 在笔者管理的环境中确实发生过此类故障，自那以后，笔者便开始使用 ARP 监控。

ARP 监控是对指定的设备发出 ARP 请求，通过是否接到回应（Reply）来进行判断。因为是通过实际进行通信来进行监控，所以漏检故障的风险很低，但即便目标主机有回应，若不能正确地对该主机发出 ARP 请求，也会导致诊断错误。为了降低漏检的风险，可通过驱动绑定对 ARP 请求目标绑定多个 IP 地址（最多 16 个）。

3.3.4 交换机的冗余

即使通过使用驱动绑定的方式绑定多块物理网卡实现了冗余，也依旧无法解决请求目标为同一交换机的交换机故障。为了避免发生交换机故障，需要准备多台交换机，并将驱动绑定的物理网卡连接到不同的交换机，这样交换机及链路就实现了如图 3.3.1 的ⓐ 所示的冗余拓扑结构 [7] 。

> [7] 在建立此拓扑结构时，可供选择的驱动绑定模式有：active-backup、balance-tlb、balance-alb。在此应用 balance-rr 及 balance-xor 会导致交换机混乱，进而导致通信被阻断。

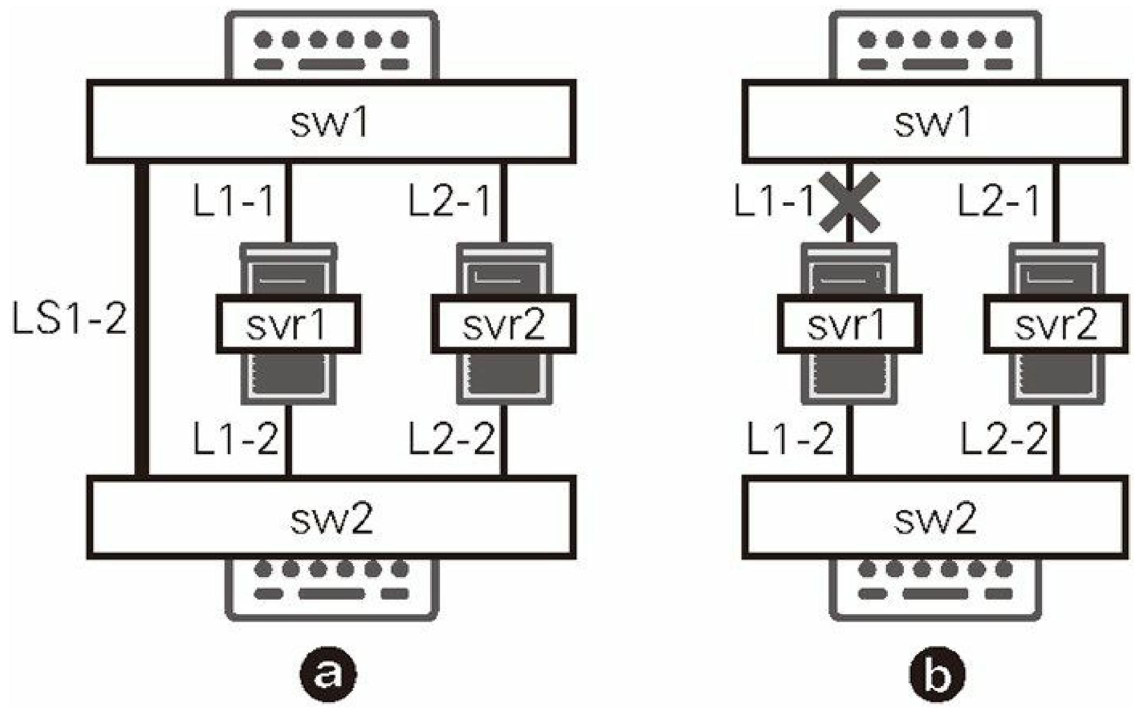


图 **3.3.1**　交换机及链路的冗余拓扑结构

成功将各个服务器设备的网线连接到其他的交换机后，便完成了驱动绑定的部署。另外需要在两台交换机之间准备 LS1-2，该 LS1-2 将在 链路故障时发挥作用。

链路故障时的行为

如果交换机之间的连接 LS1-2 发生链路故障，svr1 与 svr2 之间的通信会怎么样呢？ svr1 的 L1-1 发生故障时的情况如图 3.3.1 的\textcircled{b}所示。若需要从 svr1 发送分组（Pocket）到 svr2，由于 L1-1 无法使用，因此使用 L1-2 来进行传送。另外，因为能够确保该分组经由 sw2 传送到 svr2，因此没有发生问题。

相反，从 svr2 传送到 svr1 的分组则既有可能经由 L2-1 送出，也有可能经由 L2-2 送出。若从 L2-2 送出，就会和刚刚一样经由 sw2，则该链路可用；若从 L2-1 送出，则由于无法使用 L1-1，将无法到达 svr1。因此，为了将该分组成功传送到 sw2，必须要在路由器之间连接 LS1-2。

交换机故障时的行为

通过事先准备的 LS1-2，即便链路发生故障，也可确保 svr1 及 svr2 的运

作以使通信维持正常。但若交换机发生故障该如何是好呢？以下将探讨sw1 发生故障时的情况。

若 sw1 发生故障，则与链路故障时一样，可通过驱动绑定确认 sw1 的链路是否可用来进行判断。与链路故障不同的是，需要同时在所有服务器上进行判断。当所有服务器都不使用 sw1 的链路而只使用 sw2 的链路时，就能让通信维持正常运作。

交换机间遭遇连接故障时的情况

最后来探讨交换机与交换机之间的连接故障。在某些驱动绑定模式下，即便链路出现故障，交换机间的连接也可正常使用 [8] 。当然当故障发生时也可能造成通信阻断。应对交换机之间的连接 LS1-2 的故障时，和服务器与交换机之间的连接的情况相同，可以通过连接多根网线以实现冗余。

> [8] 例如在 balance-tlb 或 balance-alb 模式的情况下就会经常使用；在 active-backup 模式下，如果所有的服务器上被连接的 active 物理网卡与所用的交换机相同，则也有必要建立交换机与交换机之间的连接。

在服务器与交换机之间的连接尚可使用驱动绑定的方式，但交换机与交换机之间的连接方面，虽然早先可以使用交换机厂商自有的方式，但目前还是使用标准的 IEEE 802.3ad[9] 方式比较好，这常被称为“端口聚合”（Port Trunking）及“链路聚合”（Link Aggregation）。

> [9] ***URL*** http://www.ieee802.org/3/ad/

3.3.5　增设交换机

在前文图 3.3.1 的拓扑结构中，全体服务器可使用的交换机端口数实际仅为一台交换机的端口量，随着服务器台数的增加，交换机的端口数会越来越不够用，因此需要进行扩展。关于扩展的方法，既可以选择切换到目前已有的端口数富裕的交换机，也可以选择增设交换机。切换的情况和刚才介绍的没什么不同，但若要增设交换机，鉴于整个拓扑的变更，为了保持冗余性就需要满足更多的条件。对图 3.3.1 的拓扑增设交换机并进行级联连接（Cascade Connection）后，所形成的拓扑结构如图 3.3.2 所示。LS1-3 与 LS2-4 的级联连接与 LS1-2 一样，使用了链路聚合的方式实现了冗余。

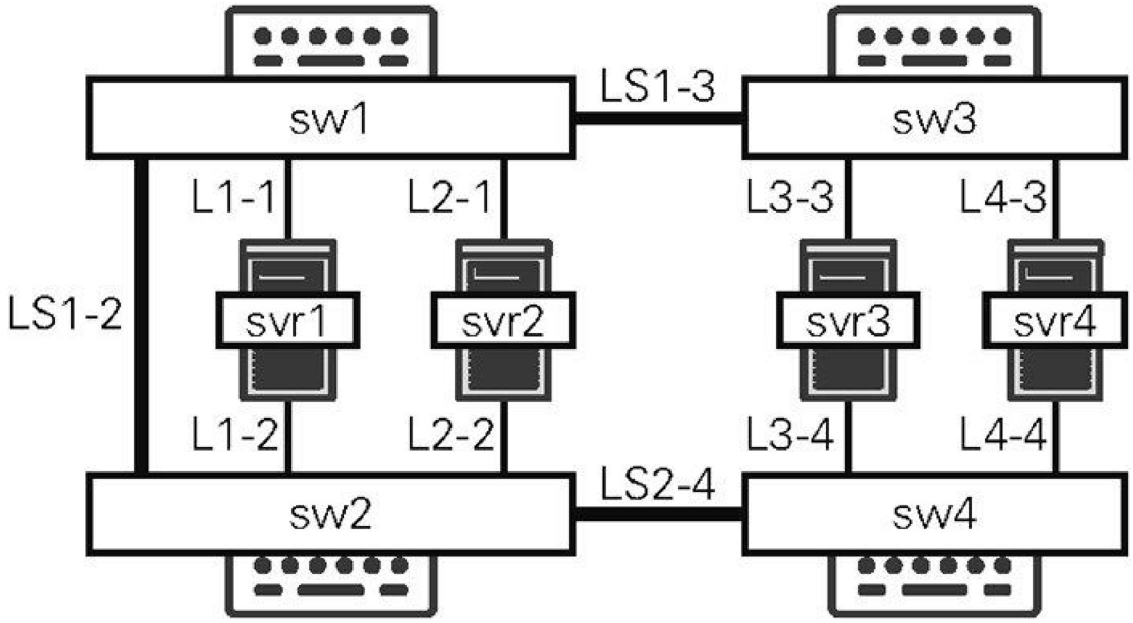


图 **3.3.2**　增设交换机并进行级联连接的拓扑结构

应用图 3.3.2 的拓扑，在驱动绑定的物理网卡监控方式下，需要使用 ARP 监控。在 MII 监控下，若 sw1 或 sw2 发生故障，则会导致 svr3 及 svr4 的通信阻断。以下具体说明该问题的状况。

若 sw1 发生故障，则 sw4 与 sw2 进行连接，sw3 被孤立。这时若 svr3 与 svr1 进行通信，因为没有 sw3 到 svr1 的线路，所以通信无法实现。若需避免该问题，在 sw1 故障时，不仅仅是 sw1 上连接的 svr1 及 svr2 到 sw3 的链路，svr3 及 svr4 到 sw3 的链路也需要被判断为发生了故障。为此，在 svr3 及 svr4 上部署物理网卡故障监控方式——ARP 监控，并将 ARP 的监控对象指定为 svr1 及 svr2 即可。这种情况下，若 sw1 发生故障，通过 L3-3 及 L4-3 发出的 ARP 请求没有得到应答，即可判断出该链路发生了故障。

实现多重冗余

在图 3.3.2 中，由于 sw3 与 sw4 之间的交换机没有连接，因此会发生 sw1 及 sw2 故障时 sw3 及 sw4 被孤立的情况。为了避免出现该问题，需要像图 3.3.3 那样在 sw3 与 sw4 之间设立一个像 LS1-2 那样的交换机间的连接 LS3-4，以设置一个将 sw3 与 sw4 连接起来的迂回线路，这样即便不使用驱动绑定的 ARP 监控，也可避免通信阻断的问题。但如果单纯这么设置迂回线路，则会导致其他问题的出现。

请看图 3.3.3，sw1 到 sw4 之间相互连接实现了环路拓扑。但若在以太网中组建该拓扑结构，将会导致广播风暴（Broadcast Storm）[10] 的出现。

[10] 广播就是指一个数据帧或分组被传输到本地网段 (由广播域定义) 上的每个节点。由于网络拓扑的设计和连接问题，或者其他某种原因，导致广播在网段内大量复制、传播数据帧，让这些广播数据充斥网络无法被处理，并占用大量网络带宽，致使网络性能下降，甚至彻底瘫痪。这就是广播风暴。——译者注

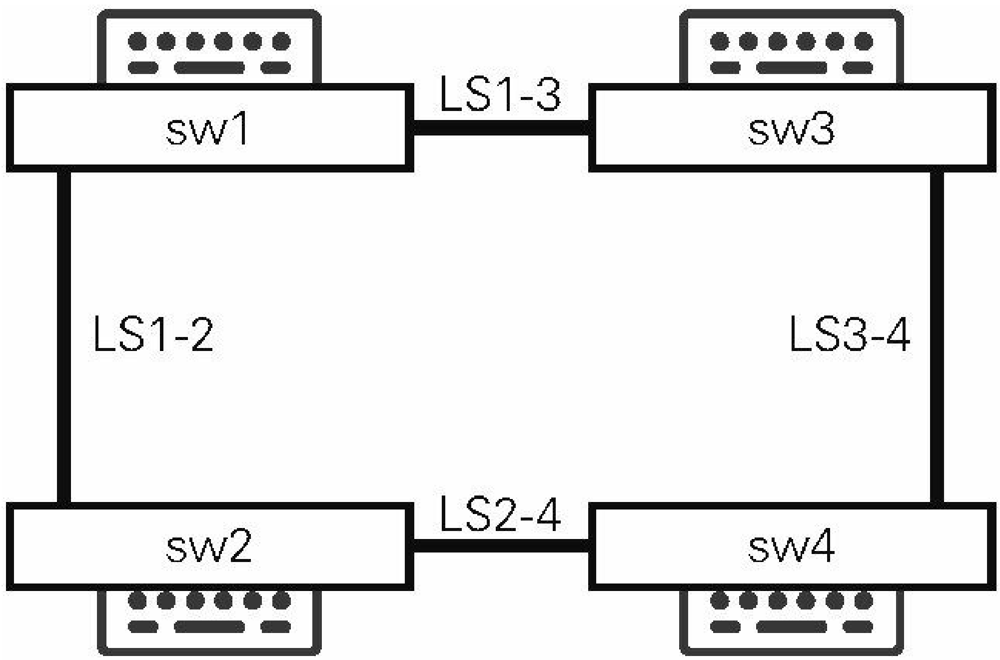


图 **3.3.3**　连接 **sw3** 与 **sw4** 的迂回线路的设定示例

如果能够确保正常情况下 sw3 及 sw4 的连接不走 LS3-4，只 在 sw1 或 LS1-3 故障时使用 LS3-4，就能够在确保迂回线路正常的前提不发生广播风暴。为了实现该拓扑结构，可使用 **STP** （Spanning Tree Protocol）。下节将讲解 STP 的加强版 RSTP 的使用方法。

3.3.6　RSTP

RSTP（快速生成树协议，Rapid Spanning Tree Protocol）[11] 是数据链路层的协议，被用于协调各个交换机，检测其环路（Loop）拓扑结构，自动屏蔽不需要的链路。RSTP 根据优先级及链路速度等设定来决定屏蔽哪些交换机的链路。

[11] 快速生成树协议是由 IEEE802.1d 定义的 STP 发展而来的。STP 在环路拓扑结构发生变化时，以 50 秒左右的时间收敛网络，而改良过的 RSTP 则可在数秒间收敛网络。RSTP 最先以 IEEE 802.1w 为标准，现在的 IEEE 802.1D-2004 中已经废止了 STP，开始使用 RSTP。

在 RSTP 中，各交换机通过相互交换 BPDU（Bridge Protocol Data Unit）消息帧，交换优先级等信息并监测故障。若该 BPDU 消息帧没有成功在交换机间进行传播，则可判断出该交换机发生了故障，即可切换到备用路线。

下面将简单介绍 RSTP 的行为方式。

网桥的优先顺序及根网桥

在 RSTP 所运作的各网桥（Bridge）[12] 中，通过相互交换 BPDU 消息帧来决定网桥的优先顺序。在链路上所有的网桥中，“根网桥”（Root Bridge）拥有最高的优先级。而其他网桥的优先顺序则是通过将从其他交换机获取的 BPDU 消息帧中所记载的数值与自身所产生的数值相比较来决定的。

[12] 网桥与交换集线器（Switching Hub）指的是同一个东西。在 RSTP 的标准用语中被称为“网桥”，因此本书中也这么称呼。

❶ 根网桥的网桥 **ID**

❷ 通向根网桥的路径开销（**Path Cost**）

❸ 一般网桥的网桥 **ID**

❹ 发送 **BPDU** 消息帧的网桥的端口 **ID**

❺ 接收 **BPDU** 消息帧的网桥的端口 **ID**

网桥 ID 是 8 个字节的数值，其中前两个字节是各网桥上用户所设定的优先级数值，后六个字节是网桥使用的 MAC 地址，网桥 ID 较小的一方优先级较高。

根网桥的路径开销是指到达根网桥所经由的链路时间，根据各个链路所设定的链路开销基数来评定。各链路的链路开销由连接接续时间决定。路径开销越小说明优先级越高。

也就是说，根网桥就是优先级数值最小的那个。

RSTP 中端口的作用

RSTP 上所有的网桥在初始化过程中，针对网桥上的各个端口，RSTP 都为其决定了特定的作用。该作用分为 5 类。

- 根端口（**Root Port**）

  在网桥的各个端口中，被接入优先级最高的网桥的端口。当 SRTP 在初始化时进行收敛，由于所有网桥所识别的根网桥相同，所以根端口就是到达根网桥的最短路径

- 指定端口（**Designated Port**）

  优先级较低的网桥所接入的端口。将链路对象的网桥端口替换为为根端口

- 替换端口（**Alternate Port**）

  除根端口外，在链路中优先级较高的端口。该端口可屏蔽除 BRDU 外的消息帧，在由于一些原因不能使用根端口的情况下，通常使用替换端口

- 备份端口（**Backup Port**）

  如果一个端口（即某个指定的端口）收到来自同一个网桥的 BPDU 消息帧，且这个 BPDU 的优先级更高，那么这个端口将成为备份端口。当两个端口被一个点到点链路的环路连在一起时，或当一个交换机存在两个或两个以上到共享局域网段的连接时，一个备份端口才能存在。由于 RSTP 所设定的网桥在接收消息时不能传送 BPDU 消息帧，所以在自身传送 BPDU 时，需要首先对没有配置 RSTP 的交换类链路的环路构成进行判断。该端口可以屏蔽除 BPDU 以外的消息帧

- 禁用端口（**Disabled Port**）

  无法接收 BPDU 消息帧的端口，通常在链路末尾将端口设为该端口

RSTP 的行为

在此将概括性地讲解 RSTP 的行为，具体请参考脚注中的资料 [13] 。

[13] 关于 RSTP 的详情，可以访问下面的链接查看 IEEE 802.1D-2004 标准。
URL http://standards.ieee.org/getieee802/802.1.html

在 RSTP（图 3.3.4）基于环路结构的网络拓扑中，通过逻辑性地屏蔽消息帧来解除环路。RSTP 中生成了“树结构”，其中的“树根”即为根网桥。在树结构中，通向优先级最高的节点的链路（= 根端口）只有一个。通常只会接入到根端口，而不会使用其他优先级较高的网桥端口，不过会事先对该端口做好标记以供无法使用根端口时使用（替换端口）。通过始终保持一个通向优先级最高的网桥链路，即可解除环路拓扑。

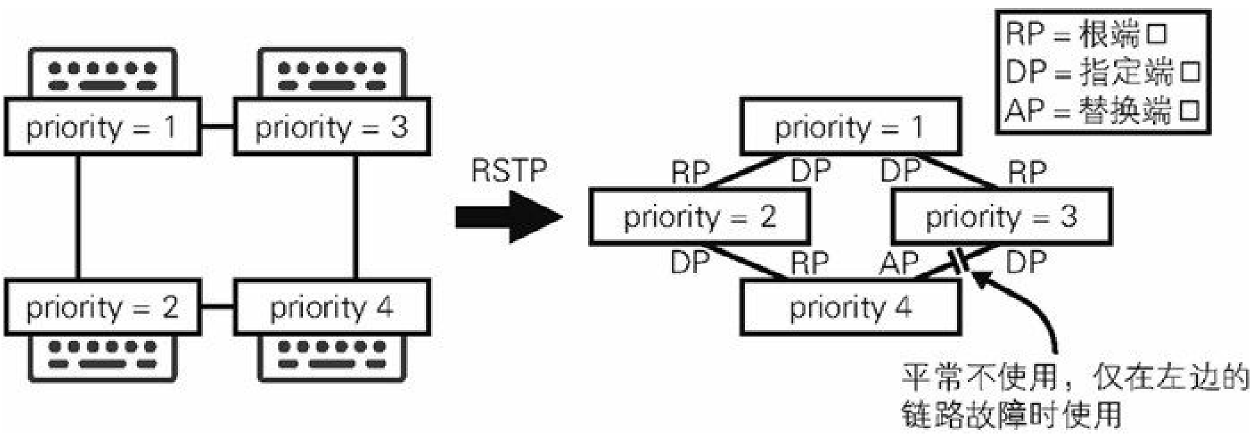


图 **3.3.4**　**RSTP** 的行为

如果网桥中没有替换端口，那么在无法使用根端口的情况下，将会与低优先级的网桥交换 BPDU 消息帧。这是因为，无法使用根端口意味着根网桥的路径开销无止尽地变大，进而导致优先级较低的网桥的优先级变高的缘故。

3.3.7　总结

使用 Linux 的驱动绑定，不仅实现了服务器与交换机之间的链路冗余，还实现了交换机的冗余。随着所设置的服务器的增多而不得不增设交换机时，若能恰当地规划拓扑结构，通过使用驱动绑定也可确保冗余性。但实施该方法的前提是连接到网络的所有设备都正确应用驱动绑定策

略。

部署 RSTP 后，即便不使用驱动绑定也可以实现交换机的冗余，这打破了以往需要在所有设备上使用驱动绑定才能实现交换机冗余的限制，争取了未来系统扩展的自由度。若链路上的服务器增加导致需要增设交换机的话，可以尝试以此为契机将 RSTP 技术应用到这些设备上。

3.4 引入 VLAN ——使网络更加灵活

3.4.1 基于服务器集群的高灵活性网络

首先请考虑何为基于服务器集群的高灵活性网络，具体来讲就是：

- 容易增加新的服务器
- 当服务器发生故障时，能立刻转移到备用机
- 某一特定功能的服务器可作为其他功能的服务器使用

只要满足了以上条件，就可以说是作业无瓶颈 = 高灵活性的网络。相反，网络拓扑结构中若存在瓶颈，无法方便地进行上述作业，则可认为该网络的灵活性较低。

硬件层面的人工操作虽然容易实现，但满足以上条件的软件实现就稍微有些难度了。例如只要服务器在身边，那么更换网线、移动服务器都不成问题。但提供 Web 服务的服务器托管在数据中心的情况下，就不能每次都让人专程跑到托管现场的数据中心进行配置。另外在物理层面更换网线时，还要小心确认机柜内的线缆情况，否则万一出现问题是很难办的。

变更网络的拓扑等需要进行物理层面的作业时，可以使用智能交换机所具备的 **VLAN** （Virtual LAN）功能，在不进行物理作业的情况下完成目标。在本节中，将要探讨基于服务器集群组建高灵活性网络的方法，即使用 VLAN 的网络拓扑结构及 VLAN 服务器集群的使用方式。另外还将专门针对负载均衡器这种特殊的硬件结构讨论 VLAN 的使用方式。

3.4.2　引入 VLAN 的优点

在服务器集群中，使用 VLAN 有众多优点。在此将探讨以下两个主要的优点：

- 使用一台交换机即可实现分段（**Segment**）管理

  → 能够灵活有效地使用交换机

  使用 VLAN，只用一台交换机就能实现分段管理。相比不使用 VLAN 的情况下，可更加灵活有效地使用交换机

- 只需合理设定，就能控制数据帧所分发的端口

  → 增加 / 更换服务器或服务器发生故障时，切换到备用机将更加容易

  通过使用 VLAN，不用在物理层面更换或修复 LAN 网线，只需通过合理的设定，就能控制数据帧所分发的 LAN 网线连接的端口。因此无论是哪个段连接的服务器发生故障，只需将 LAN 线缆接入备用机，就能很容易地修复该故障

下面我们就来详细看一下实际的运行情况。

交换机的有效使用

在如图 3.4.1 那样的普通的 Web 系统中，存在 WAN 段及内网段两部分。

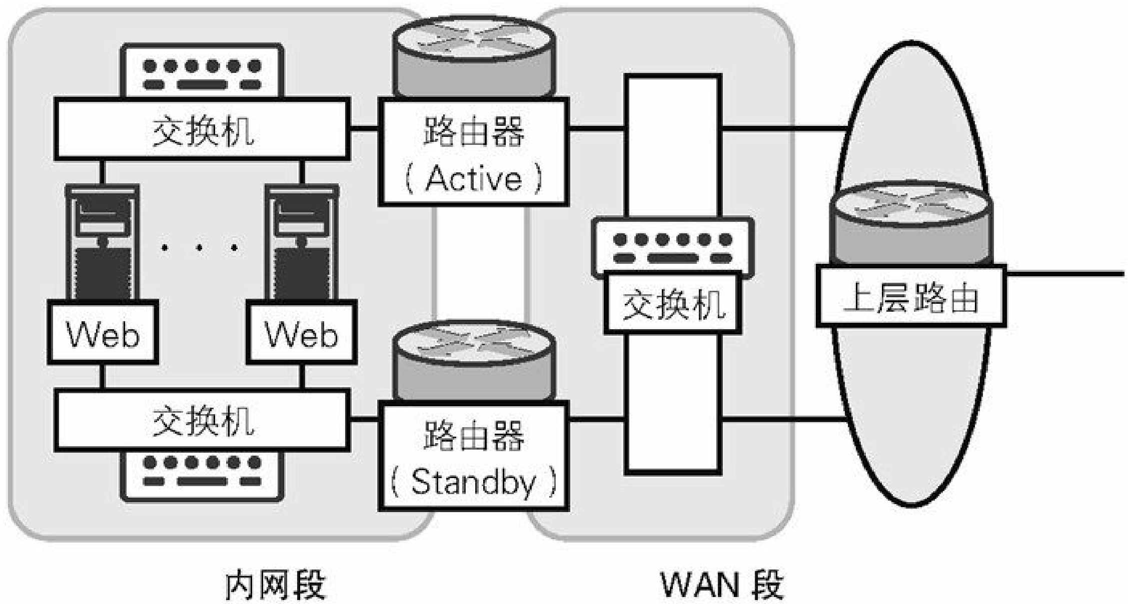


图 **3.4.1**　普通的 **Web** 系统

下面来看 WAN 段，即便除交换机以外的端口在拓扑中占用了相当多的资源，交换机也最多只能使用四个端口（上层拓扑两个，路由器两个）。就算使用八口交换机，也有四个端口是被浪费掉的。

服务在逐渐扩展时肯定需要增加 Web 服务器的数量，若内网段的交换机没有存在空闲端口，即便增加一两台 Web 服务器，也需要相应地增设交换机。但此时 WAN 段的端口数又稍显充裕，若能在内网段的交换机上使用这些空闲端口，则就没必要增设新的交换机了。

当然，若直接将外部网络与内部网络接入到同一台交换机上，就会造成外网也能直接访问内网的风险。因此不能只追求弹性化，还要考虑到相应的安全问题。

通过在以上拓扑结构中使用 VLAN，即可在确保安全的同时，灵活使用各个交换机的端口。

故障服务器的恢复机制 **...**灵活使用一台备用机

在此假设数据中心中某台服务器发生了故障，让我们来考虑如何恢复这台服务器。简单来说，如果数据中心已经准备好了一台恢复用的备用机，那么就需进行以下步骤：

- 将故障服务器的设置迁移到备用机
- 接入备用机以取代故障服务器

为了缩短恢复故障所花费的时间，事先将相关服务器设置迁移到备用机上或许是个不错的方法。但由此就需要事先将 Web 服务器、数据库服务器等每个生产环境中的系统设置都迁移到备用机上，当工作中的服务器多达上百台时，将需要非常庞大的系统规模，因此现实中该方案无法实施。

没有发生故障时，尚未使用的备用机产生的费用开销也很大，因此需要尽可能地减少备用机的数量。在包含网关或 Linux 内核的负载均衡器中，可以部署一台备用机，使服务器、网关及负载均衡器都能将其作为备用机使用。

使用 **VLAN**，仅用一台备用机就能实现恢复

接下来将以上述仅用一台备用机来保护整个系统的拓扑结构为例，探讨故障时的恢复方法。首先考虑上述 ❶ 中将环境设置迁移到备用机的方法 [14]。

> [14] 除这些方法外，还有其他可供选择的方案。但由于那些方案并不对应网络层的问题，因此在本节中不做探讨。

- 需要将存放数据的多个磁盘安装到备用机上，并在运行时进行切换
- 将目标服务器的系统结构同步，并在启动后启动必要的服务，通过赋予指定的 **IP** 地址进行切换
- 网络启动系统拓扑中的这些服务器，启动时需要对服务器的参数进行合理设置 [15]

> [15] 有关网络启动的详情请参考 5.5 节。

以下将探讨上述 ❷ 中接入备用机的问题，具体如何实现呢？可以在备用机上安装多块网卡，以便可以连接到任意网络上。但这样做就需要交换机的所有段都接入到代替机上，也就是说，从物理层面进行人工配置

以及网线的妥善处理等是很难实现的。

为了应对该问题，可以不将备用机设为特别的拓扑结构，无论备用机连接哪个交换机的哪个端口，只要能让备用机接入就可以了。若能实现这种理想环境，根据情况将正在运行的服务器紧急分配于其他用途也是可以的。

有关网线的连接操作，可以灵活使用交换机及系统中的 VLAN 功能。通过运用 VLAN，就可以缓解必须人工进行物理层面的操作的限制，在非故障时增设服务器也变得更加容易，使系统逐渐晋升为大规模高并发的拓扑结构成为了可能。

通过实施上文中描述的操作，就满足了本节开头所介绍的高灵活性的网络需求。下文将结合该用途对 VLAN 进行详细讲解。

3.4.3　VLAN 的基础知识

VLAN 并非是物理层面的结构，而是通过网络设备及服务器的设定来“逻辑性”地分割拓扑结构的技术。具体来说，广播域（Roadcast Domain）的分割从理论上讲是完全可能的。通过使用 VLAN，即使同一台交换机连接多个段的终端，根据设定的不同，也可以将广播域进行逻辑性分割，从而只将数据帧转发（Forward）到适当的端口。

上文中已经针对使用 VLAN 逻辑性地分割网络进行了说明，但仍需考虑需要接入到不同网络的两台交换机是如何通过一台交换机进行管理的。例如像图 3.4.2 的 ❶ 那样，在不同的段——集群 1 及集群 2 所分发的系统中，通常需要在每个集群中都准备交换机。但若恰当地使用 VLAN 进行设定，即可实现如图 3.4.2 的 ❷ 那样仅用一台交换机，就能够分割出多个集群。

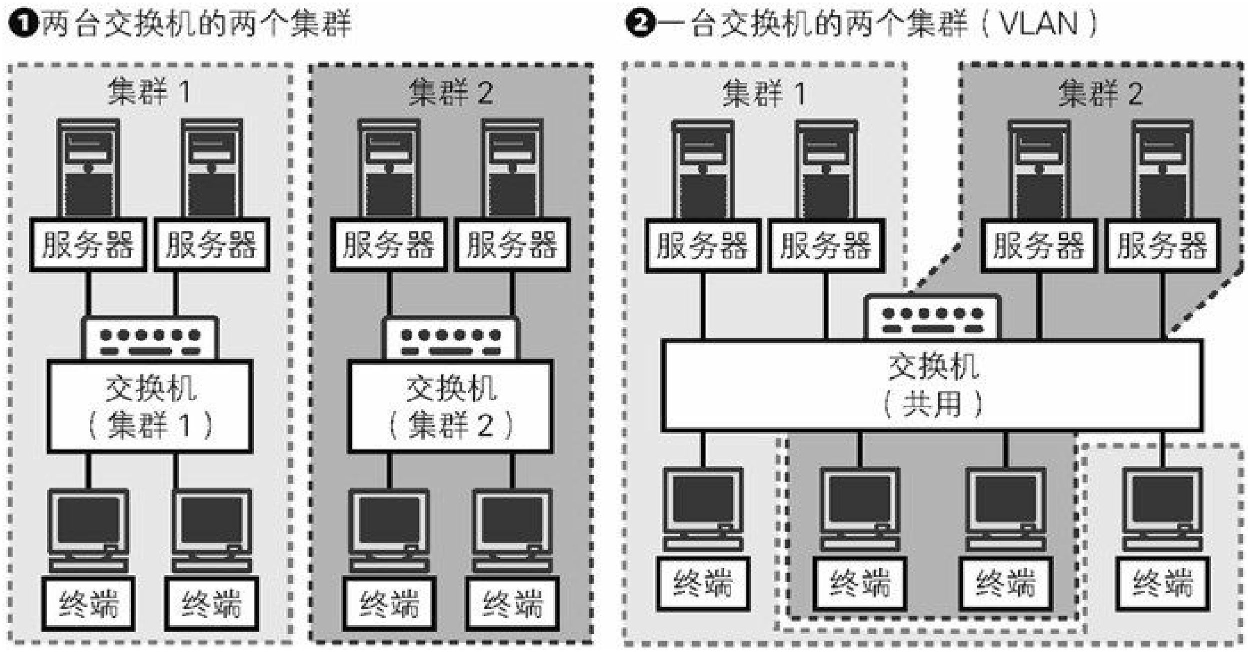


图 **3.4.2**　网络物理层面的分割及逻辑层面的分割（**VLAN**）

在不使用 VLAN 的情况下，若一个交换机接入了不同的段，该交换机也可针对这些不同的段分别进行通信。但在尚未设定 VLAN 的交换机中，多播（Multicast）、广播帧（Broadcast Frames）或地址不详的未知单播帧（Unknown Unicast Frame）就会被转发到毫无关系的段上。

在此本来就没有关系的段的广播帧等也会被转发到各个端口，从而造成毫无意义的带宽浪费。根据情况，在本来就无法进行通信的段上发送通信信息，会带来盗听等危险，因此还需要同时留意安全问题才行。

3.4.4　VLAN 的种类

VLAN 的实现大体可分为两种方法。

其中之一是手动将集群分配到每个端口上，即“静态 VLAN”（Static VLAN）。使用该方法时，若端口所连接的终端集群发生改变，则需要手动变更路由器上的相关设置。

还有一种是根据所连接的设备等，动态分配集群的“动态 VLAN”（Dynamic VLAN）。使用该方法时，即便所连接的集群终端需要改变，也无需在路由器上手动进行变更，因为分配会基于规则动态地改变。

实现 VLAN 技术的方法多种多样，根据使用目的的不同使用方法也会不同。其实近些年来在公司内部的 LAN 上，通常会根据用户或 MAC 地址等信息制定相关的规则，以分配合适的 VLAN 集群来确保安全。在公司内部 LAN 中，由于使用者（= 职员）的调动会造成拓扑频繁地发生改变，而通过 VLAN 的动态分配，可以让使用者获得便利的同时也满足系统安全的需要，因此该方法被广泛使用。

但由于使用服务器集群不用频繁地改变拓扑结构，所以无需考虑有关此类改变的便捷性。但若频繁地发生结构变化，则需要重新考量整个拓扑结构的合理性。

在 VLAN 中，还存在上述一些厂商为自己的硬件定制的特殊种类，这些特殊种类上文中没有介绍到。下面将以服务器集群中常用的 VLAN 知识为准，对“端口 VLAN”及“标记 VLAN”进行介绍。

端口 **VLAN**

端口 VLAN（Port VLAN）是为交换机的各个端口分配 VLAN ID 的方法。一个端口对应一个 VLAN ID，在集群上的端口同样具备相应的识别 ID。在图 3.4.3 中，端口 1、5、6 为 VLAN1（VLAN ID 1 所分配的集群），端口 2、3、4 为 VLAN2（VLAN ID 2 所分配的集群）。在该拓扑结构下，同一 VLAN 之间自然能实现通信，但 VLAN1-VLAN2 之间则无法实现通信。

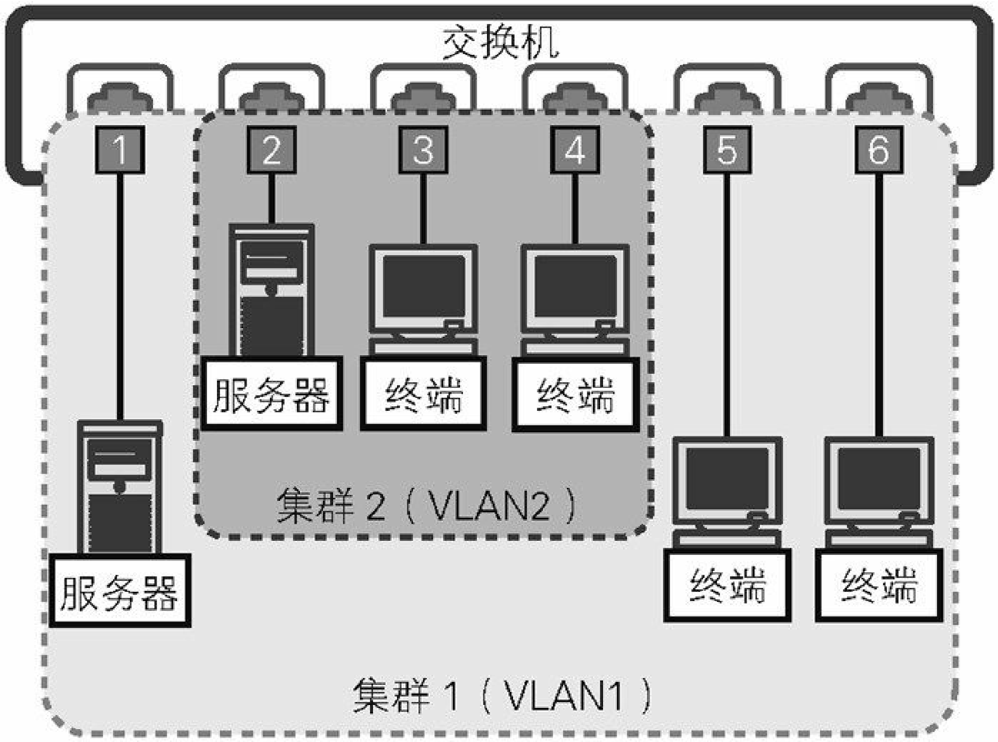


图 **3.4.3**　端口 **VLAN**

使用端口 VLAN 的优点是，连接到该交换机的客户端不需要特别的设置，此外因为是针对每个端口的 VLAN 设置，可以在一台交换机上完成，因此结构较为简单。但使用端口 VLAN 也存在缺点，比如无法在多台交换机进行集群化分组。使用端口 VLAN，可以在多台交换 机间进行集群操作，但结果会导致拓扑结构变得更加复杂，交换机间的流量也会被这些毫无关系端口消耗等，因此不建议实施。在端口 VLAN 的拓扑结构中，需要将大规模数据交换的服务器想办法接入到同一个交换机上。

标记 **VLAN**

标记 VLAN（Tagged VLAN）是指，将包含 VLAN ID 的 VLAN 识别信息（以下简称 VLAN 标签）标记到以太网帧（Ethernet Frame）中，以作为识别各 VLAN 集群的方式。该 VLAN 标签是在客户端或交换机发出以太网帧的时机被插入的。在 VLAN 集群的管理上，与以端口为单位进行集群操作的端口 VLAN 不同，其是以所传输的数据帧为单位进行集群化分组的。因此就没有了端口 VLAN 中“一个端口对应一个 VLAN ID”的限制，使得单位端口也可针对多个 VLAN 进行操作。

通过该方式就能简单地构成如图 3.4.4 那样的横跨多个交换机的 VLAN 集群拓扑结构。

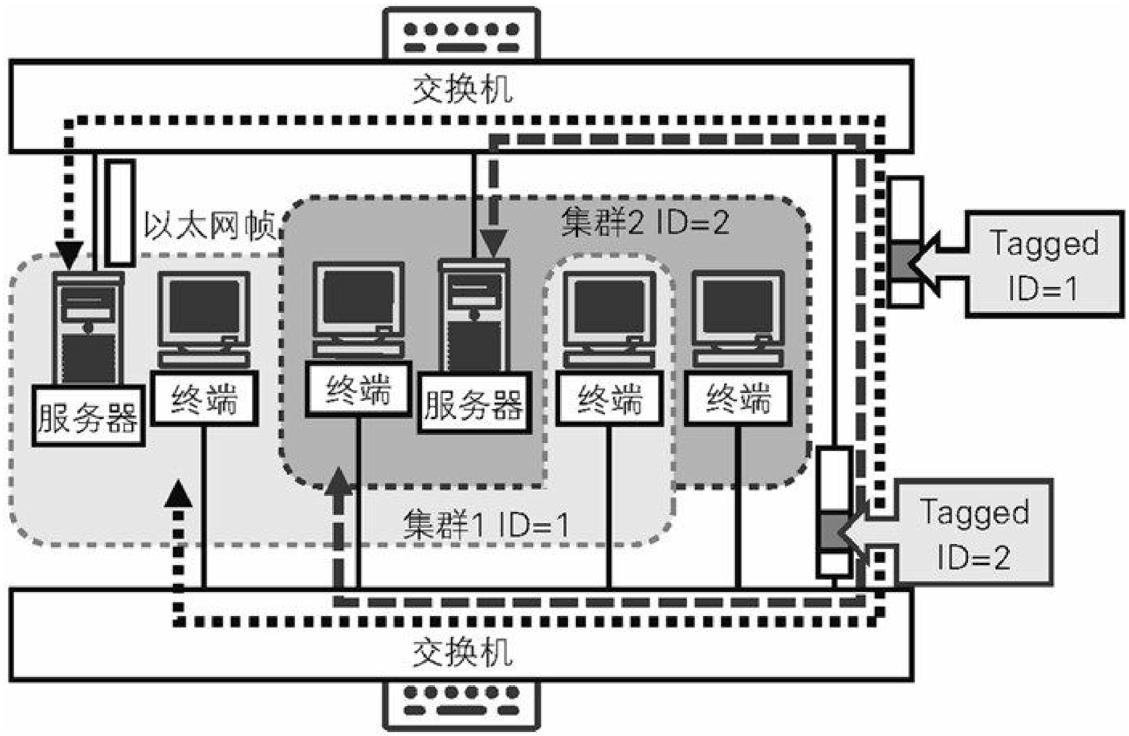


图 **3.4.4**　标记 **VLAN** 标签

至此好像一直在说标记 VLAN 相对端口 VLAN 的优势，但其实附有 VLAN 标签的以太网帧（被标记的以太网帧）与一般的以太网帧相比，报文首部不同了。因此从不能识别 VLAN 标签的终端或同样不能识别 VLAN 标签的网络设备看来，这是不正确的以太网帧。也就是说，如果将附带 VLAN 标签的以太网帧转发到这些设备，结果可能会导致将本来有效的数据帧舍弃了，因此需要确保终端或网络设备能够正确识别这些附带有 VLAN 标签的以太网帧。

3.4.5　在服务器集群中的使用

接下来以 3.4.2 节中介绍的服务器故障的应对为前提，讨论使用各 VLAN 技术来组建拓扑结构。以下将考虑包含负载均衡器在内的基于 Linux 核心的服务器系统拓扑结构。

不使用 **VLAN** 的拓扑结构

首先来考虑不使用 VLAN 的拓扑结构。如图 3.4.5 所示，在实现整个冗余的拓扑结构中，需要面对下面几个问题：

- **WAN** 端的交换机只有 **4** 个端口可用（其余的端口被浪费）
- 准备的备用机没有接入到 **WAN** 端的交换机时，不能作为负载均衡器的备用机使用
- 由于负载均衡器的存在造成硬件结构较为特殊（需要 **4** 块网卡）

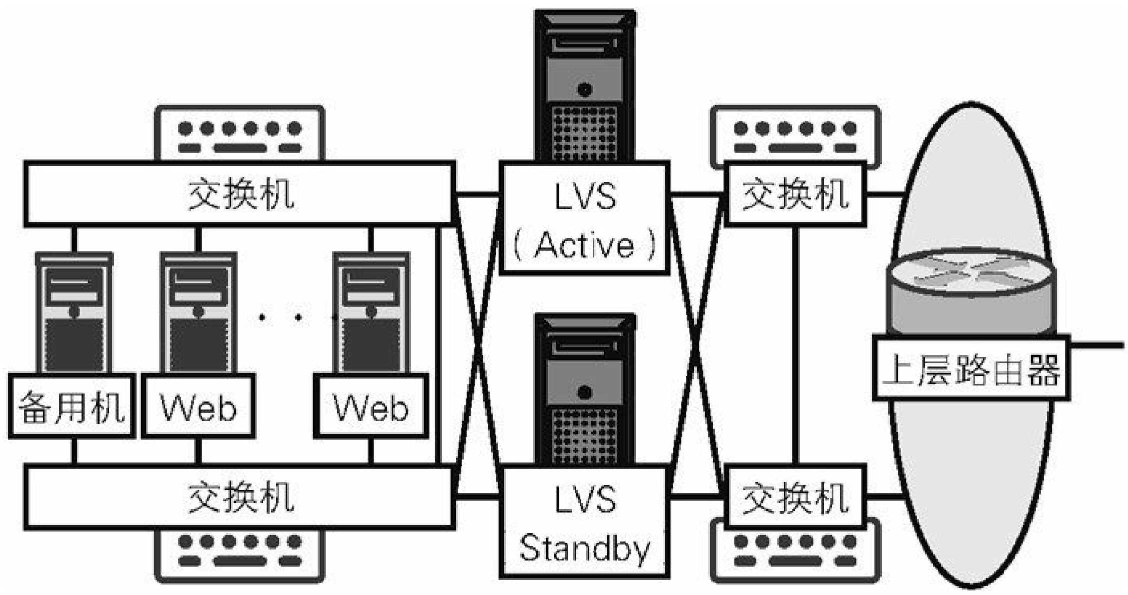


图 **3.4.5**　整个冗余的拓扑结构

为了改善以上问题，可以考虑如图 3.4.6 那样将所有连接都接入到交换机上的拓扑结构。

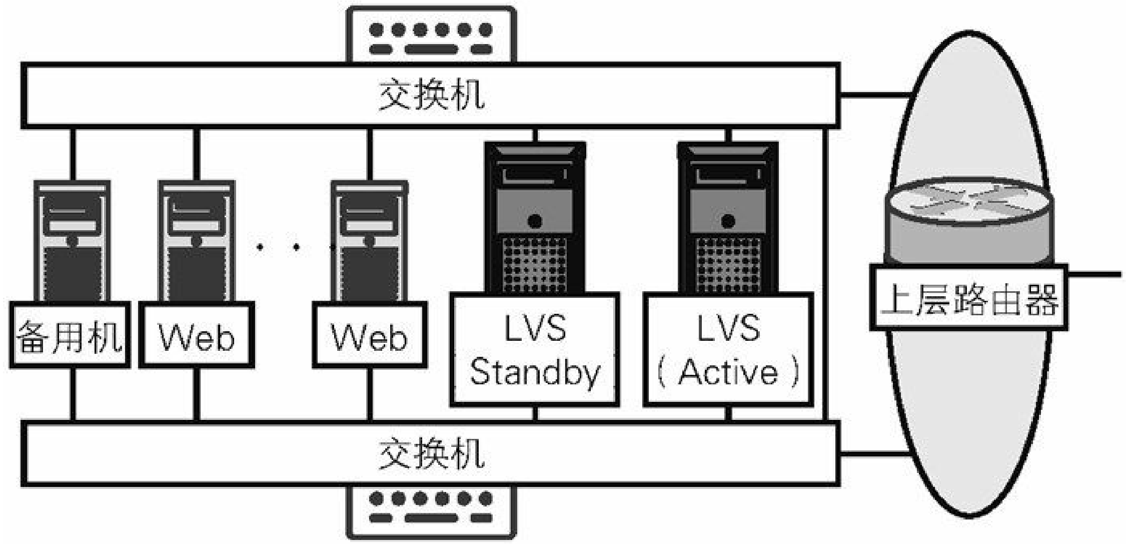


图 **3.4.6**　接入同一台交换机的全冗余拓扑结构

但该方式存在一个重大的问题。如前所述，多播 / 广播帧会被转发到所有的端口上。也就是说，该结构会将 Web 服务器段所发生的多播 / 广播帧传送到 WAN 段的上层路由器。相反 WAN 段也同样会接到该数据帧。因此无论是从流量上来说还是从安全方面来说都是不理想的。也就是说如果不解决这个问题，图 3.4.5 的拓扑结构就依然存在缺陷。

使用端口 **VLAN** 的拓扑结构

使用端口 VLAN 的情况下会是什么样的拓扑结构呢？

请看图 3.4.7 的使用端口 VLAN 的结构。同一个交换机内有两个 VLAN 集群（VLAN ID1、VLAN ID2），分别为 WAN 段及内部段所使用，因此就能更有效地利用交换机的端口了。

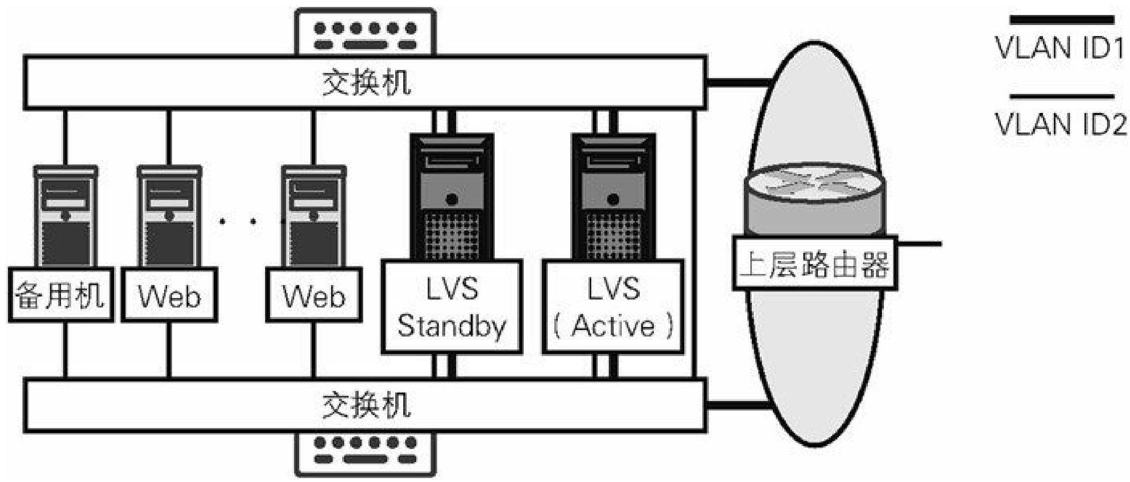


图 **3.4.7**　使用端口 **VLAN** 的拓扑结构示例 **1**

虽然通过改变连接备用机端口的 VLAN 集群，即可切换到 WAN 段，但由于负载均衡器是拥有 4 块网卡的特殊硬件结构，因此考虑到负载均衡器可能会发生故障，为了能够将备用机当作负载均衡器使用，需要在备用机上也安装 4 块网卡。此外，还必须考虑到如图 3.4.8 的多个交换机结构中交换机之间的连接问题。由于该问题的存在，无法单纯使用 VLAN 构建出理想的拓扑结构，实乃遗憾。

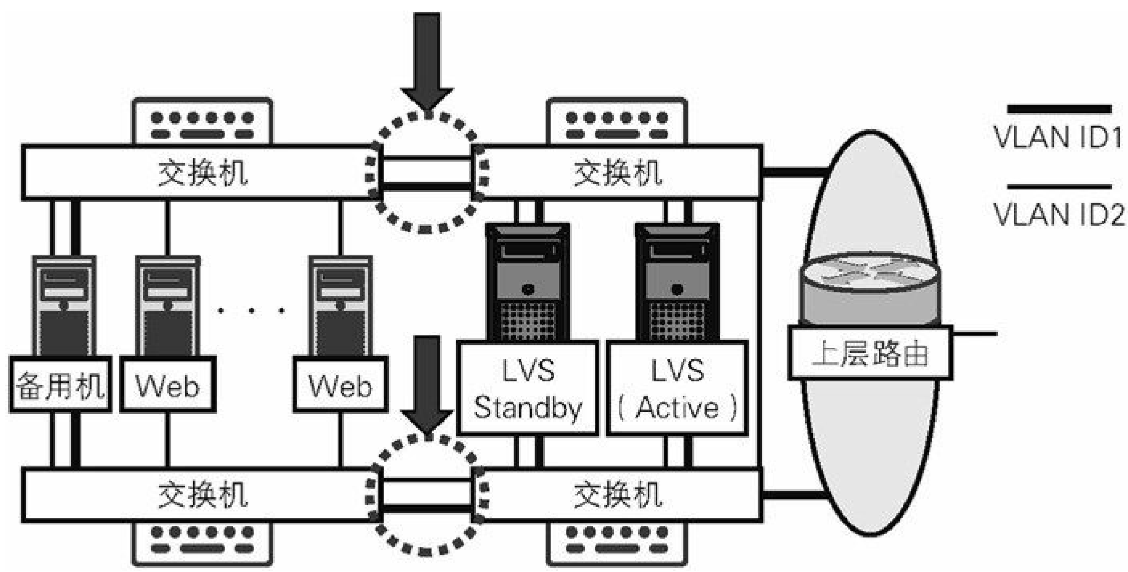


图 **3.4.8**　使用端口 **VLAN** 的拓扑结构示例 **2**

使用 **VLAN** 标签的拓扑结构

终于轮到探讨使用 VLAN 标签的情况了。

针对使用端口 VLAN 的拓扑结构所存在的两点问题来考虑改善方案，可得到如图 3.4.9 的结构。首先将连接到负载均衡器端口的 VLAN ID1 与 VLAN ID2 两个集群进行必要的配置以实现可操作，另外准备两块负载均衡器的网卡，这方面与备用机及其他服务器一样，像这样组建的话，问题就得到了解决。

在本结构下，集群中所接收的数据帧均包含 VLAN 标签。如前所述，若终端无法识别包含 VLAN 标签的以太网帧，那么该数据帧就很有可能不被处理而直接被忽略。但实际上，终端连接存在限制的标记 VLAN，与基本上可连接任何终端的端口 VLAN 这两种方式是可以混合使用的。也就是说并不需要为所有的数据帧都附带 VLAN 标签，仅在端口 VLAN 处理较为困难时，也就是必须要处理多个 VLAN 的情况下，通过标记 VLAN 标签进行处理即可。

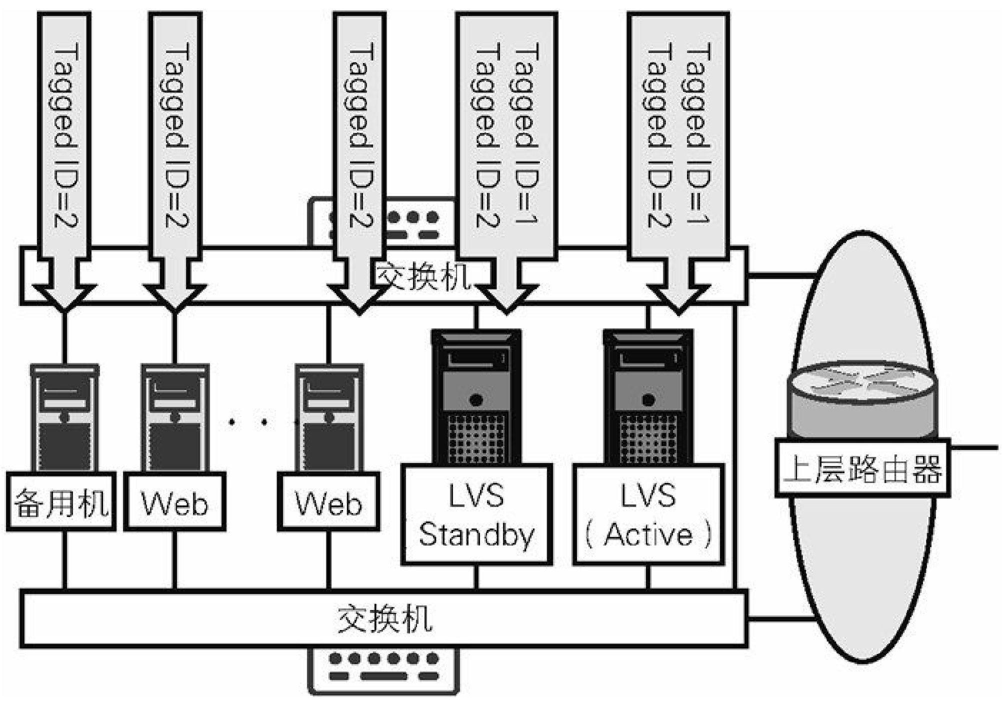


图 **3.4.9**　使用 **VLAN** 标签的拓扑结构

在这种情况下，若首先将连接到上层路由器的端口分配到 VLAN ID1 的端口上，剩下的端口即 VLAN ID2 也就只能设定为端口 VLAN 了。另外，将多个 VLAN 集群的数据帧交换所必经的端口（负载均衡器的端口），设定为“有”VLAN 标签，即将负载均衡器的端口 VLAN ID1 设定

为“有”VLAN 标签，反之，将 VLAN ID1 设定为端口 VLAN，并 将 VLAN ID2 设定为“有”VLAN 标签也是可以的。另外，通过事先设定“当 VLAN 使用频繁时，不使用标记 VLAN 标签的形式，而使用端口 VLAN”的规则，可以更简单地避免使用时的混乱。

通过以上设定，能够处理 VLAN 标签的终端就只有负载均衡器了，所以在此需要特别对负载均衡器进行说明。因为 Linux 支持标记 VLAN，因此不需要准备特别的硬件，只需进行配置就可以了。在 Linux 上使用标记 VLAN 的情况下，需要内核支持及相应的配置工具。要确保内核支持，只需在编译时设定“CONFIG_VLAN_8021Q=y ”，就能开启该功能；另外配置工具方面可以使用vconfig 命令，使用该命令可建立 VLAN 接口，当然该工具也能撤销该接口的建立。

在本拓扑结构中的负载均衡器上执行 vconfig ，就建立了 VLAN ID1 的新端口 eth0.1。

```
lvs01:~# aptitude install vlan
lvs01:~# vconfig add eth0 1
```

该拓扑结构对不使用 VLAN 或只使用端口 VLAN 时的问题做出了相当程度的改善。

通过使用标记 VLAN，能够更容易地搭建出类似的物理结构，让整个系统具备相当大程度的伸缩性。但在这里需要注意一个问题，即使用标记 VLAN 的情况下逻辑结构往往会变得相对复杂。而且与使用端口 VLAN 时不同，可能还需要增加对服务器的设定。因此建议将端口 VLAN 作为基本的结构，只在必要的时候采用标记 VLAN 的设定。

3.4.6 即便在复杂的 **VLAN** 结构下，也需要让物理层面的设备结构尽可能简易化

上文的一系列说明都在阐述在服务器集群中使用 VLAN 的优点。

在实际引入 VLAN 时最重要的是，无论使用哪种 VLAN 方法，都应该确保拓扑结构相对简易。在耗费精力引入 VLAN 技术后，如果因结构

复杂而使故障的解决很花时间，甚至引起了别的故障等，那就太得不偿失了。

而且虽说要注重逻辑方面的结构，但也不能忽视物理层面的结构。正如在存在横跨多个交换机的段的情况下，该段的数据需要在这些交换机之间传送，如果此类段有很多，就可能存在带宽瓶颈（图 3.4.10）。为了避免出现该问题，在初期规划拓扑结构时就应该对物理结构组态及逻辑结构进行合理规划。根据需要，也可以考虑引入链路聚合等确保带宽的技术。

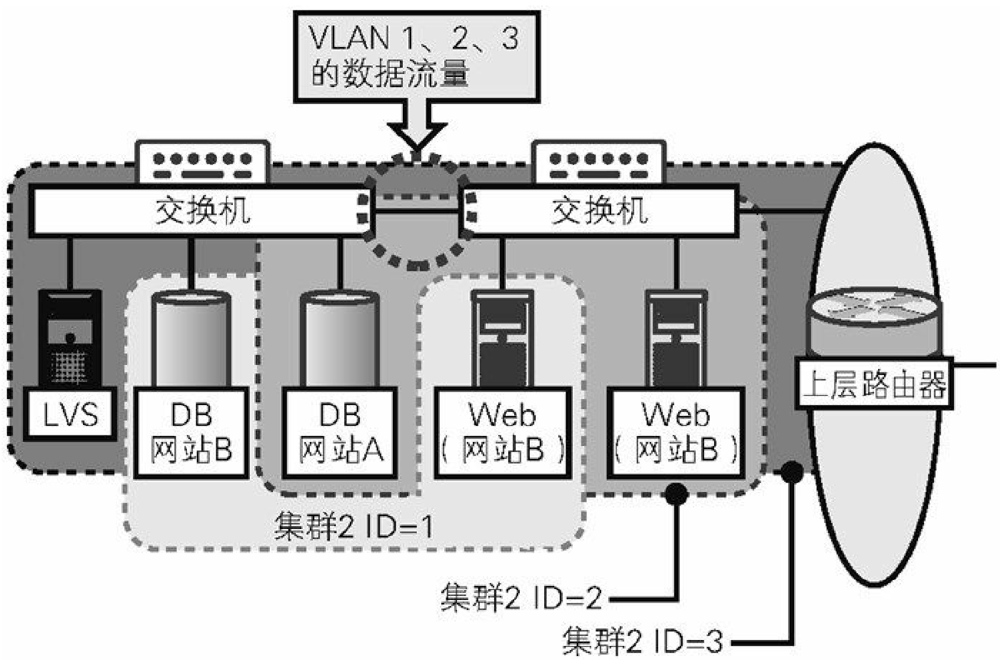


图 **3.4.10**　交换机之间出现带宽瓶颈的可能性

第 4 章　性能优化、调整——Linux单个主机、Apache、MySQL

4.1　基于 Linux 单个主机的负载评估

4.1.1　充分发挥单个主机的性能

提起“负载分流”，大多数情况下人们都会联想到将“负载”“分流”到多台服务器进行处理的机制，前三章所叙述的也是这种情况。

但首先需要明确的是，将原本一台服务器就能负载处理的内容，分流到十几台服务器，是否显得有些本末倒置呢？不首先想办法充分发挥单个服务器的性能，而跳过需要的优化步骤分流到多台服务器进行负载均衡，这样做意义就不大了。所以本章的首要问题，就是针对构成该网络的“单个主机”做重点讲解。

清楚何为性能、何为负载

为了充分发挥单个主机的性能，需要首先了解“什么是性能”。为此就需要掌握服务器资源的使用状况，因此接下来将首先对相应的监控方式进行说明。在讲解监控方式的同时，也将顺便指出 Linux 操作系统中相关对象的运行机制。所谓知其负载必先知其操作系统之机制。而如果不知道操作系统的运作行为，就难以诊断出系统当前的状态。

参考 Linux 的内核源码（Kernel Source），就能具体明白“负载”的定义了。虽说通过监控可以得知状态信息，但不了解监控的实质含义就无法进行考察衡量。例如需要在多任务处理（Multitasking）的运行原理中明白进程状态和负载之间的关系。Web 应用程序的负载分发通常都是“分流或减轻磁盘 I/O”的工作。不同操作系统中 I/O 的优化策略也有所不同。操作系统为了减轻 I/O 设立了缓存等机制，充分利用系统中的缓存是降低系统 I/O 的诀窍。

如果知道操作系统的运行原理和负载的监控方式，原本只能治标，现在也能从根本上解决了。若操作系统出现瓶颈，系统出现了性能问题，也

可以按照基本策略查明其原因。下面我们来看一下具体方法。

针对单一主机实施优化策略意味着需要确认在操作系统上运行的中间件。虽说只要操作系统以理想状态运行，应用软件基本上都能发挥出相应性能。但世事难料，服务器不能保证 100% 不出问题。所以接下来我们也将了解一下 Web 服务器和数据库服务器的优化操作，这是 Web 程序 + 数据库软件运作的核心。

本章是以 Linux 2.6 版内核的操作系统为基础来介绍的。但由于近几年多任务处理的操作系统也基本上都以类似的原理运作，因此即使是其他版本的内核或操作系统，本章的介绍也能适用。

4.1.2 别臆断，请监控

发挥单个主机的性能需要正确掌握服务器资源的使用情况。也就是说，需要监控服务器的负载究竟到了什么样的程度，因此这一监控工作对降低单个服务器的负载也是非常重要的。

在程序员之间有一句很有名的格言：

别臆断，请监控

负载分流的情况也不例外。

因为 Apache 和 MySQL 等应用软件都运行于操作系统上，所以监控负载情况就意味着要监控操作系统。若对操作系统都不了解，何谈负载分流？操作系统（此处为 Linux 内核）可以查看负载所需的全部信息。

在 Linux 中使用`ps`、`top`、`sar` 等工具。这些工具都是 Linux 命令行工具，可以据此检查并监控 Linux 内核内部的各种统计信息。例如，运行`top` 命令后，终端如图 4.1.1 所示。

```
top - 19:50:21 up 150 days,  4:38,  1 user,  load average: 0.70, 0.66, 0.59
Tasks: 104 total,   2 running, 102 sleeping,   0 stopped,   0 zombie
Cpu(s): 21.8%us,  0.6%sy,  0.0%ni, 77.2%id,  0.0%wa,  0.1%hi,  0.3%si,  0.0
Mem:   4028676k total,  2331860k used,  1696816k free,   150476k buffers
Swap:  2048276k total,     9284k used,  2038992k free,   425064k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
18481 apache    17   0  394m 154m 4228 R  100  3.9   1:23.50 httpd
```

```
19199 apache    16   0  390m 153m 4328 S   26  3.9   0:41.59 httpd
18474 apache    15   0  360m 122m 4364 S    4  3.1   1:12.26 httpd
18471 apache    15   0  371m 133m 4232 S    2  3.4   1:13.31 httpd
19325 apache    15   0  343m 105m 4340 S    2  2.7   0:35.10 httpd
    1 root      15   0 10304   80   48 S    0  0.0   4:23.65 init
    2 root      RT   0     0    0    0 S    0  0.0   2:40.25 migration/0
    3 root      34  19     0    0    0 S    0  0.0   0:00.38 ksoftirqd/0
<以下省略>
```

图 **4.1.1**　**top** 的输出示例

`top` 命令是表示系统瞬间状况的快照信息。该工具所输出的信息会随着时间流逝刷新内容，因此非常便于观察系统负载的趋势。

例如，使用`top` 工具能够查阅系统的 CPU 使用率和内存的使用状况等数值。通过从这些数值判断，虽然能清楚地确认系统的负载状况，但是这些数值涉及的项目及类型很多，究竟如何了解这些数值的意义呢？为此就需要清楚“什么是负载”。

在详细介绍“什么是负载”之前，我们先来看一下如何确认生产环境中发生的瓶颈。

4.1.3　确认瓶颈的基本流程

要想清楚了解生产环境中所发生的瓶颈的情况，请注意以下几点：

- 观察 **load average**（平均负载）
- 观察 **CPU**、**I/O** 是否存在瓶颈

接下来将详细讲解各个基本流程。

观察 **load average**

首先，作为重要的负载评估指标，需要通过 `top` 和 `uptime` 等命令查看 load average。load average 可以衡量整个系统的负载情况。虽说仅通过 load average 还不能判断出系统出现瓶颈的原因在哪里，但可从 load

average 着手进行系统瓶颈的调查。

有时在 load average 较低时，系统的吞吐量有可能还是达不到要求。此时就需要查看软件的配置是否不合理，网络、远程主机方面是否存在问题等。

观察 **CPU**、**I/O** 是否存在瓶颈

当 load average 较高时，需要确认 CPU 和 I/O 哪个存在问题。通过使用`sar`或`vmstat`等工具，可以确认一段时间内的 CPU 使用率和 I/O 等待的情况。确认后即可进入下一个阶段。

当 **CPU** 负载较高时

CPU 负载较高时，可以按照下面的流程来排查问题所在：

- 通过 **top** 和 **sar** 等工具来判断是用户程序（**Web** 应用程序）的原因，还是系统程序（操作系统）的原因
- 一边通过 **ps** 命令来查看进程状态以及 **CPU** 使用时间等数据，一边锁定出现问题的进程
- 如果要进一步了解所锁定的进程的具体行为，可以使用 **strace** 命令进行跟踪（**Trace**），也可以使用 **oprofile** 工具在计数器中断处理入口处建立监测点，以缩小监控瓶颈的范围

造成 CPU 负载过重的原因通常是：

- 磁盘及内存占用尚未超标的情况下（即 **I/O** 不存在瓶颈），系统的 **load average** 指标依旧很高
- 程序的逻辑部分负载超过 **CPU** 的承受能力且不受控制

如果是前者且系统吞吐性能也出现了问题，可以尝试通过增设服务器、改善程序逻辑和算法来解决问题。后者的情况下要注意程序没有预料到的流程，以避免出现程序失控的状况。

当 **I/O** 负载较高时

I/O 负载较高基本上有两种情况：一种是来自程序本身的磁盘读写较为频繁，造成该程序输入输出负载比较高；另一种是因使用了 Linux SWAP 交换区而造成磁盘的频繁访问。通过使用 `sar` 及 `vmstat` 命令确认 SWAP 交换区的状况，即可分辨出到底是哪一种情况。

如果确认后发现发生了频繁访问 SWAP 交换区的情况，可通过以下几个步骤来查出问题所在：

- 通过 **ps** 命令确认是否有特定进程消耗了大量内存
- 由于程序不受控制而造成内存占用过大时，可以修改相应的程序逻辑来解决问题
- 系统内存不足时增加内存。无法增加内存的情况下需要考虑进行负载分流

如果没有频繁访问 SWAP 交换区，且磁盘上依然频繁发生输入输出，该情况下可以认为是缓存所需的内存不足。可以根据服务器装载的磁盘容量以及可以扩充的内存容量，分以下几种情况来考虑：

- 通过扩展内存缓存以实现扩展缓存容量时，优先扩充内存容量
- 无法通过扩充内存容量来解决问题时，需要考虑数据分流和增设缓存服务器等方案。当然也可以考虑改善程序以减轻 **I/O** 频率

以上是寻找发生负载的原因的基本方法。在此基础上，接下来就让我们一起来了解下“为什么发生瓶颈后，逐步缩小造成系统瓶颈问题的原因的范围是完全可以实现的”。

4.1.4　何为负载

究竟负载指的是什么呢？针对负载，下文将从以下几点着手介绍。

两种负载

一般来说，负载可以分为两大部分：

- **CPU** 负载

- **I/O** 负载

例如，假设有一个进行大规模科学计算的程序，虽然该程序不会频繁地从磁盘输入输出，但是处理完成需要相当长的时间。因为该程序主要被用来做计算、逻辑判断等处理，所以程序的处理速度主要依赖于 CPU 的计算速度。此类 CPU 负载的程序称为“计算密集型程序”（CPU Bound）。

还有另一类程序，主要从磁盘保存的大量数据中搜索找出任意文件。这个搜索程序的处理速度并不依赖于 CPU，而是依赖于磁盘的读取速度，也就是输入输出（Input/Output，I/O）。磁盘越快，检索花费的时间就越短。此类 I/O 负载的程序，称为“I/O 密集型程序”（I/O bound）。

一般来说，AP 服务器所做的处理是将从数据库服务器获得的数据进行加工后交给客户端。在此过程中，该 AP 服务器应该不会发生大规模的 I/O，因此通常可认为 AP 服务器是计算密集型的服务器。

另一方面，数据库服务器是构成 Web 应用程序的另一系统要素，其主要工作是从磁盘检索数据，特别是当数据规模较大时，比起 CPU 的计算时间，I/O 速度更能影响数据库服务器的快慢，因此数据库服务器属于 I/O 密集型服务器。另外，即使是相同的服务器，根据造成负载的原因不同，其特性也会发生很大变化。

多任务操作系统与负载

Windows 和 Linux 等近几年的多任务操作系统能够同时处理几个不同名称的任务。但是在同时运行多个任务的过程中，CPU 和磁盘这些有限的硬件资源就需要被这些任务程序共享。即在很短的时间间隔内，需要一边在这些任务之间进行切换一边进行处理，这就是多任务（图 4.1.2）。

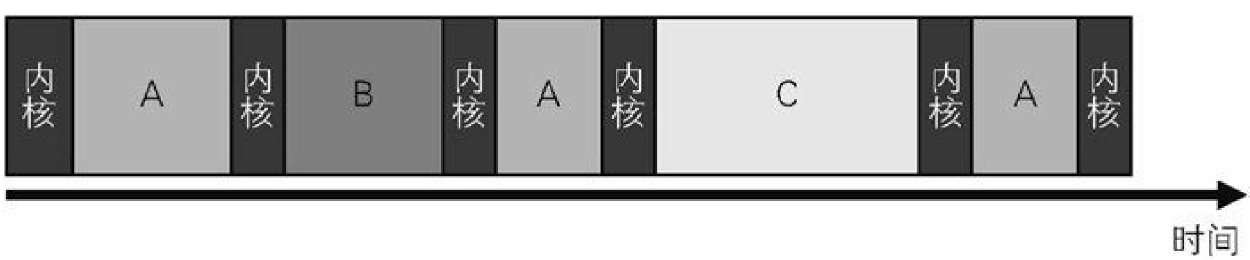


图 **4.1.2**　多任务

运行中的任务较少的情况下，系统并不等待此类切换动作的发生。但是当任务增加时，例如任务 A 正在 CPU 上执行计算，接下来如果任务 B 和 C 也想进行计算，那么就需要等待 CPU 空闲。也就是说，即使想运行处理某任务，也要等到轮到它时才能运行，此类等待状态就表现为程序运行延迟。

`top` 的输出中包含“load average”的数字。

```
load average: 0.70, 0.66, 0.59
```

load average 从左边起依次是在过去 1 分钟、5 分钟、15 分钟内，单位时间的等待任务数，也就是表示平均有多少任务正处于等待状态。在 load average 较高的情况下，就说明等待运行的任务较多，因此轮到该任务运行的等待时间就会出现较大延迟，即反映了此时负载较高。

但是，load average 的数字只是表示等待的任务数，仅根据 load average 并不能判断具体是 CPU 负载高还是 I/O 负载高。因此最终要判断服务器资源具体哪里遭遇了瓶颈，还需要再做一些细致的调查。

- 具体看哪个数值能判断系统的瓶颈呢？各个数值分别反应了系统哪里的状态呢

- **load average** 表示的**“**等待任务**”**具体等待的是什么呢

- 即使找到了瓶颈，也需要深入了解具体是哪个进程负载的原因所导致的瓶颈

搞清负载的本来面目 = 知晓内核的行为

所谓“负载”，其实也只不过是用一个词表示了因多任务的服务器资源争夺而产生的等待时间。为了搞清负载的本来面目，需要明白系统在什么情况下会进行“等待任务”的动作，也就是需要明白 Linux 内核的运行。

Linux 内核中的“进程调度器”（Process Scheduler）是支配任务等待的程序。在多任务的支配中，进程调度器负责决定任务运行的优先级，以及让任务等待或使之重新开始等内核中的核心工作。通过进程调度器查看

进程的概要，从而就可以掌握该进程所对应的负载情况。

进程调度和进程状态

“进程”（Process）是程序在系统中运行的基本单位。进程与表示内核中的运行的单位“任务”（Task）在狭义上虽然有所区别，但这里把两者理解为一个意思也不妨碍对下文的理解。

例如，在运行 `ls` 命令的时候，`ls` 的二进制文件的机器语言命令在内存中被展开，紧接着 CPU 在内存中访问（Fetch）命令并执行。为了完成该命令的执行，需要了解`ls` 命令使用的各个内存区域的地址、运行中的命令位置（程序计数器，Program Counter）、`ls` 命令打开的文件一览等各种信息。通过将这些信息归纳整理，汇总出正在运行的程序，可以更方便地弄清楚状况。因此，此处所指的进程，即汇总了“程序命令”和“运行时所需信息”的对象。

在 Linux 内核中，每一个进程都存在一个名为“进程描述符”（Process Descriptor）的管理表。该进程描述符中保存有各种执行时的信息 [1] 。

> [1] 进程描述符在 Linux 内核编码中采用 task_struct 结构。具体在内核源代码的 include/linux/sched.h 中有其定义，有兴趣的读者可以研究一下。

在 Linux 内核中，各进程描述符会被调整为按照优先级降序排列，以按合理的顺序运行进程（任务）。这个调整即为“进程调度器”的工作（图 4.1.3）。

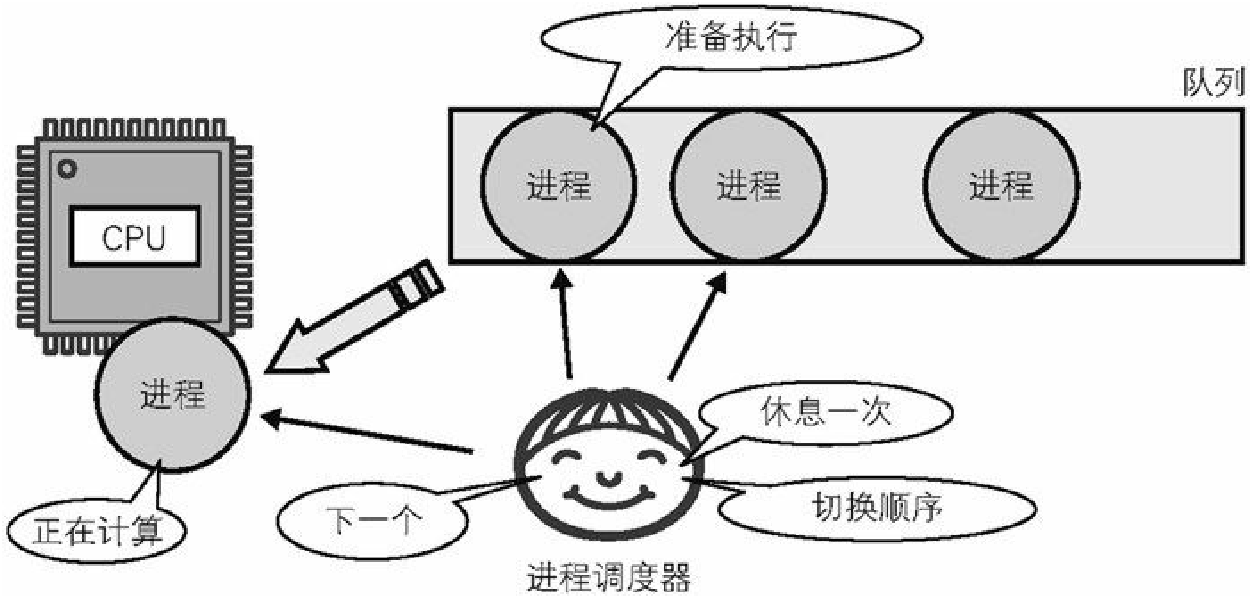


图 **4.1.3**　进程调度器

调度器划分并管理进程的状态。例如，

- 等待分配 **CPU** 资源的状态
- 等待磁盘输入输出完毕的状态

进程描述符中有保存此状态的区域（task_struct 结构体中的 state 成员）。各个状态的区别如表 4.1.1 所示。

表 **4.1.1**　进程描述符的状态的区别

状态	说明
TASK_RUNNING	运行状态。只要CPU空闲，任何时候都可运行
TASK_INTERRUPTIBLE	可中断的等待状态。主要为恢复时间无法预测的长时间等待状态。例如系统睡眠或来自于用户的输入的等待等
TASK_UNINTERRUPTIBLE	不可中断的等待状态。主要为短时间恢复时的等待状态。例如磁盘输入输出的等待

TASK_STOPPED	响应暂停信号而运行中断的状态。直到恢复（Resume）前都不会被调度
TASK_ZOMBIE	僵死状态。虽然子进程已经终止（exit），但父进程（进程控制块）尚未执行wait()，因此该进程的资源没有被系统释放

调度器在管理各个进程的运行状态的同时，还会根据情况变更状态以控制任务的执行顺序。

进程状态变迁的具体例子

下面来看个具体的例子。该例子中有三个进程Ⓐ 、Ⓑ 、Ⓒ 同时运行。首先，每个进程在生成后都是可运行状态，也就是从 TASK_RUNNING 的状态开始。这里请注意 TASK_RUNNING 是“可运行的等待状态”，而不是“现在正在运行”的状态。

- 进程Ⓐ：**TASK_RUNNING**
- 进程Ⓑ：**TASK_RUNNING**
- 进程Ⓒ：**TASK_RUNNING**

TASK_RUNNING 的三个进程立即成为调度对象。此时，假设调度器给进程Ⓐ：TASK_RUNNING分配了 CPU 的运行权限。

- 进程Ⓐ：**TASK_RUNNING** 正在运行
- 进程Ⓑ：**TASK_RUNNING**
- 进程Ⓒ：**TASK_RUNNING**

由于在 Linux 内核中无法区别正在运行的状态和可运行的等待状态。为了直观起见，这里将该状态称为“TASK_RUNNING 正在运行”。

因为分配了 CPU，所以进程Ⓐ 开始处理。进程Ⓑ 和Ⓒ 则在此等待进程

Ⓐ 迁出 CPU。

假设Ⓐ 进行若干计算之后，需要从磁盘读取数据。那么在Ⓐ 发出读取磁盘数据的请求之后，到请求的数据到达之前，将不进行任何工作。此状况称为“Ⓐ 因等待 I/O 操作结束而被阻塞”。在 I/O 完成处理前，Ⓐ 一直处于等待状态（TASK_UNINTERRUPTIBLE），并不使用 CPU。于是调度器就查看Ⓑ 和Ⓒ 的优先级计算结果，将 CPU 运行权限交予优先级较高的一方。

这里假设Ⓑ 的优先级高于Ⓒ 。

- 进程Ⓐ：**TASK_UNINTERRUPTIBLE**
- 进程Ⓑ：**TASK_RUNNING** 正在运行
- 进程Ⓒ：**TASK_RUNNING**

Ⓑ 刚开始运行，就需要等待用户的键盘输入。于是Ⓑ 进入等待用户输入状态，同样被阻塞。结果就变成Ⓐ 和Ⓑ 都要等待输出，运行Ⓒ 。这时Ⓐ 和Ⓑ 都是等待状态，但是等待磁盘输入输出和等待键盘输入为不同的状态。等待键盘输入是无期限长时间的事件等待（TASK_INTERRUPTIBLE），而读盘则是必须短时间内完成的事件等待，这是两种状态有所区别的原因。

各进程的状态如下所示。

- 进程Ⓐ：**TASK_UNINTERRUPTIBLE**（等待磁盘输入输出 **/** 不可中断）
- 进程Ⓑ：**TASK_INTERRUPTIBLE**（等待键盘输入 **/** 可中断）
- 进程Ⓒ：**TASK_RUNNING** 正在运行

这次假设在进程Ⓒ 运行的过程中，进程Ⓐ 请求的数据从磁盘到达了缓冲装置。紧接着硬盘对内核发出中断信号，内核知道磁盘读取完成，将进程Ⓐ 恢复为可运行状态。

- 进程Ⓐ：**TASK_RUNNING**

- 进程Ⓑ：**TASK_INTERRUPTIBLE**
- 进程Ⓒ：**TASK_RUNNING** 正在运行

此后进程Ⓒ 也会变为某种等待状态。例如：

- **CPU** 的占用时间超出了上限
- 任务结束
- 进入 **I/O** 等待

一旦满足这些条件，调度器就可以完成从进程Ⓒ 到进程Ⓐ 的进程状态的切换。

进程状态迁移汇总

以上进程的状态变迁如图 4.1.4 所示。像这样进程被定义为几个状态以作区分，进程在各状态间迁移的同时会进行必要的计算或者改变响应 I/O 的行为。进程的状态变迁对深入理解系统负载有很大的意义。

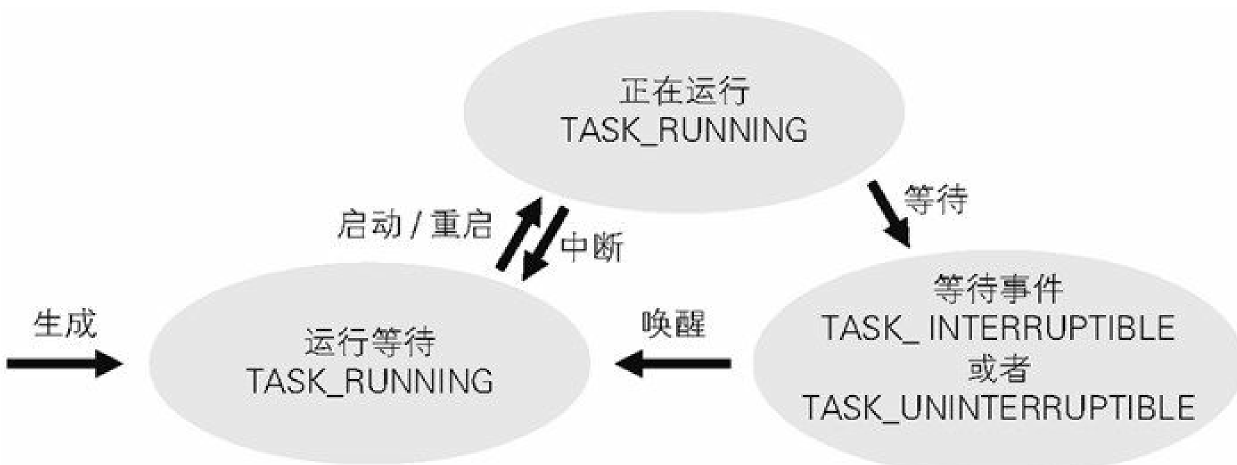


图 **4.1.4**　进程的状态迁移

换算到 **load average** 的等待状态

进程 A~C 在发生状态迁移的过程中，出现了四种状态：

- **TASK_RUNNING** 正在运行
- **TASK_RUNNING**
- **TASK_INTERRUPTIBLE**
- **TASK_UNINTERRUPTIBLE**

load average 是表示“等待进程的平均数”的数字。四个状态中，除“TASK_RUNNING 正在运行”以外，其余三个都是等候状态。而这三个进程的等待状态都会被加权换算为相应的 load average 吗？

事实证明，只有 **TASK_RUNNING** 和 **TASK_UNINTERRUPTIBLE** 会被换算为 load average，TASK_INTERRUPTIBLE 则不能被换算。也就是说：

- 即便需要即刻使用 **CPU**，也还需等待其他进程用完 **CPU**
- 即便需要继续处理，也必须等待磁盘输入输出完成才能进行

以上两种情况的进程才会表现为 load average 的数值。

两种状态的共同点是“虽然需要即刻运行处理，但是无论如何都必须等候”。另一方面，即使同样是在等待，键盘输入等待或睡眠等待与程序明示的等待也不相同，它们不包含在 load average 换算的范围中。另外，来自远程主机的数据，例如来信等待，由于无从得知对方何时会发来数据，因此也不能换算为 load average。

load average 是表示系统负载的指标，可见以上两点的等待状态与系统负载是挂钩的。

load average 表述的负载意义

硬件会以特定的周期间隔向 CPU 发出中断信号。因为是按特定的间隔发出的信号，因此被称作“计时器中断”（Timer Interrupt）。例如在 CentOS 5 中，中断间隔被设置为 4ms（毫秒）。每次中断时，CPU 都将推进时间，计算该时间间隔内运行的进程所占用的 CPU 情况等，进行与时间有关的处理。由此在每个计时器中断的间隔中，计算出该时间间隔内 load average 的数值。

在内核的计时器中断中，可以预先算出处于可运行状态的任务和处于 I/O 等待状态的任务的数量。通过用该数字除以单位时间，就可以算出相应的 load average。

*　　*　　*

到目前为止，“load average 显示的负载的本来面目”就显而易见了。总之，load average 所描述的负载就是：

需要运行处理但又必须等待队列前的进程处理完成的进程个数。

具体来说就是：

- 等待被授予 **CPU** 运行权限的进程
- 等待磁盘 **I/O** 完成的进程

这个确实与直观感觉相符。在很占用 CPU 资源的处理中，例如在进行动画编码等的过程中，虽然想进行其他相同类型的处理，结果系统反应却变得很慢，还有从磁盘读取大量数据时，系统的反应同样也会变得很慢。但另一方面，无论有多少等待键盘输入操作的进程，也不会让系统响应变得很慢。

专栏

用工具观察进程状态**……ps**

虽然TASK_RUNNING和TASK_INTERRUPTIBLE等在内核中的处理状态存在区别，但是可以从用户进程查看此状态。具体可以通过 ps 和 top 等命令将该信息以一定的格式显示出来。以下是 ps 命令的输出。

```
% ps auxw | egrep (STAT|httpd)
USER       PID %CPU %MEM    VSZ   RSS TTY   STAT START   TIME COMMAND
root     10861  0.0  1.7 295256 69020 ?     Ss   Feb07   0:06 /usr/sbin/h
apache   18711  7.2  3.1 366744 125176 ?    R    00:13   1:14 /usr/sbin/h
apache   18827  8.1  3.8 396636 154696 ?    S    00:18   0:58 /usr/sbin/h
apache   18898  9.0  3.9 400188 158492 ?    S    00:22   0:42 /usr/sbin/h
```

请注意STAT列。根据man ps的结果，“S”是“Interruptible Sleep”，相当于TASK_INTERRUPTIBLE。“R”是“Running or Runnable（onrunqueue）”※1 ,相当于TASK_RUNNING※2 。可见 ps 的STAT项和内核的进程状态是对应的关系。

※1　runqueue即运行队列，处于可运行状态的进程在内核中排列的队列。

※2　“Ss”的s表示控制进程（Session Leader）。

- R(Run):TASK_RUNNING
- S(Sleep):TASK_INTERRUPTIBLE
- D(Disk Sleep):TASK_UNINTERRUPTIBLE
- Z(Zombie):TASK_ZOMBIE

其他项目所表示的意义请参照ps的参考手册。

4.1.5　计算 **load average** 的内核编码

为了让大家对 load average 的计算方式能了解得更加具体，接下来我们来研究一下它在内核中的代码。这里参照 Linux 内核 2.6.23 的编码。

计算 load average 的函数为 kernei/timer.c 的 **calc-load()** 。每次硬件产生计时器中断时都会调用该函数。在 CentOS 5 中，计时器中断的周期是 4ms，因此每 4ms 就会调用一次 calc-load()。

```
unsigned long avenrun[3];
EXPORT_SYMBOL(avenrun);

static inline void calc_load

(unsigned long ticks)
{
    unsigned long active_tasks; /* fixed-point */
    static int count = LOAD_FREQ;

    count -= ticks;
    if (unlikely(count < 0)) {
        active_tasks = count_active_tasks
```

```
();
        do {
            CALC_LOAD(avenrun[0], EXP_1, active_tasks);
            CALC_LOAD(avenrun[1], EXP_5, active_tasks);
            CALC_LOAD(avenrun[2], EXP_15, active_tasks);
            count += LOAD_FREQ;
        } while (count < 0);
    }
}
```

可以看出，在 calc-load() 函数中，全局数组 averun 中存储了 **count_active_tasks()** 函数的计算结果。count_active_tasks()，顾名思义，就是统计该时间系统内存在的“Active 的任务（进程）数”。但这个“Active 的任务”究竟是什么呢？如果再试着追踪处理流程，就会发现是 kernel/sched.c 的 **nr_active()** 函数。

```
unsigned long nr_active

(void)
{
    unsigned long i, running = 0, uninterruptible = 0;

    for_each_online_cpu(i) {
        running += cpu_rq(i)->nr_running;
        uninterruptible += cpu_rq(i)->nr_uninterruptible;
    }

    if (unlikely((long)uninterruptible < 0))
        uninterruptible = 0;

    return running + uninterruptible;
}
```

for_each_online_cpu() 枚举了 cpuid（即枚举了系统当前使用所有 CPU 核心的 ID 值）。还有，**cpu_rq()** 是取得 CPU 相关运行队列 [2] 的宏。因此在这里可以清楚地按顺序查看各 CPU 的运行队列。在运行队列中，

> [2] 存储等待状态的任务描述符的队列。

- **cpu_rq(i)->nr_running**
- **cpu_rq(i)->nr_uninterruptible**

获取以上两个数值并进行计算，然后返回结果。从名字就能看出，二者分别相当于各运行队列内的 TASK_RUNNING 和 TASK_UNINTERRUPTIBLE 的进程数。

这个 nr_active() 所返回的数值会被传递给前述的 **calc_load()** 函数，并以 1 分钟、5 分钟、15 分钟为单位来进行计算，计算得出的值将存储在 averun 数组中，因此 averun 数组中所存储的数值就是 load average 的原形。

如果用户进程发出了读取 proc 文件系统的 **/proc/loadavg** 的请求，内核就会将请求时的 averun 数组加以整理并发送至用户空间。让我们先来确认一下输出。

```
% cat /proc/loadavg
0.01 0.05 0.00 4/46 10511
```

`top` 或`uptime` 命令从该输出中取得 load average，并将其显示出来。

在本节中，我们了解了在系统源码中，每次计时器中断都会进行 load average 的计算，即代码将通过统计 TASK_RUNNING 及 TASK_UNINTERRUPTIBLE 状态的等待任务数来计算出 load average。

4.1.6 通过 **load average** 判断 **CPU** 使用率和 **I/O** 等待时间

通过观察 load average 具体的计算方法，就可以明白该数值表示了 CPU 负载和 I/O 负载。例如系统负荷过重的情况下，大多数也是因为 CPU 或 I/O 出现了问题。因此，查看 load average 后发现需要有所应对时，接下来就要调查 CPU 或 I/O 哪里有问题。

使用 **sar** 来查看 **CPU** 使用率及 **I/O** 等待时间

通常在系统中，CPU 使用率和 I/O 等候时间 (I/O 等待率) 等指标都是不断在变化的，可以通过 `sar` 命令来确认这些指标。正如 `sar` （System Activity Reporter）的全称所表述的那样，该工具常被用于浏览系统状况报告。该工具包含在 sysstat 软件包内。

图 4.1.5 是计算密集型服务器系统中 sar 的运行结果。

```
% sar
Linux 2.6.19.2-103.hatena.centos5 (jubuichi.hatena.ne.jp)      02/08/08

00:00:01          CPU     %user     %nice   %system   %iowait    %steal
00:10:01          all     59.84      0.00      1.54      0.00      0.00
00:20:02          all     48.72      0.00      1.48      0.00      0.00
00:30:01          all     54.91      0.00      1.45      0.00      0.00
00:40:01          all     66.39      0.00      1.51      0.02      0.00
Average:          all     57.47      0.00      1.49      0.01      0.00
```

图 **4.1.5**　**sar** 的运行实例（计算密集型服务器系统）

详细的用法我们稍后再做讲述。sar 优于其他工具的一点是，可以随着时间的流逝来浏览并比较负载指标。例如在上述 00:00~00:40 时间段中，CPU 使用率的变迁就可以通过sar 来确认。“%user”是用户程序的 CPU 使用率，“%system”是系统所占用的 CPU 使用率。当 load average 较高且这些 CPU 使用率的数值也同样较高的情况下，就可以断定造成等待进程负载的原因是 CPU 资源不足。

CPU 的用户模式和系统模式

CPU 的用户模式和系统模式分别是：

- 用户模式：用户程序运作时的 **CPU** 模式，也就是一般的应用软件运作模式
- 系统模式：系统程序即内核运作时的 **CPU** 模式

用户及系统占用 CPU 资源是分开来表述的。

通常在应用程序的 CPU 负载较为严峻的情况下，用户模式的 CPU 使用率会变高。也就是说，用户应用程序处于正在进行计算的状态。另一方面，例如运行大量进程和线程时，进程及线程的切换次数较多，或者系统调用较为频繁时，系统模式的 CPU 使用率会变高。

用户模式及系统模式的 CPU 占用行为的不同如图 4.1.6 所示。

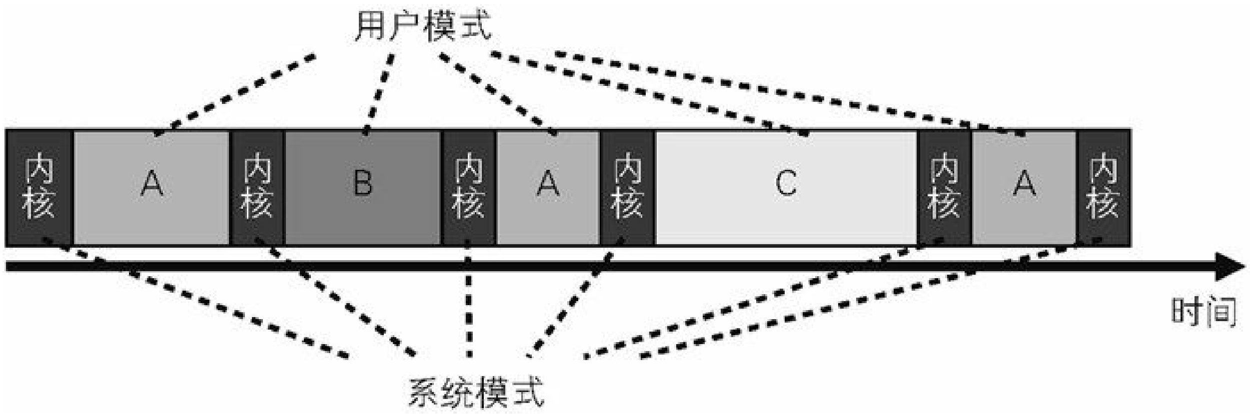


图 **4.1.6　CPU** 时间的不同

正如本章开头叙述的那样，多任务的实现方式就是内核在短时间内切换进程。也就是说，在切换进程时，内核必定处于运作状态。这时如果系统发生调用，运行状态就会从用户程序变迁到内核。

I/O 密集型服务器的 **sar**

接下来是 I/O 密集型服务器中的 sar 的结果（图 4.1.7）。

```
Linux 2.6.18-8.1.8.el5 (takehira.hatena.ne.jp)  02/08/08

00:00:01        CPU     %user     %nice   %system   %iowait    %steal     %idle
00:10:01        all      0.14      0.00     17.22     22.88      0.00     59.76
00:20:01        all      0.15      0.00     16.00     22.84      0.00     61.01
00:30:01        all      0.16      0.00     19.66     18.99      0.00     61.19
00:40:01        all      0.10      0.00      8.50     13.09      0.00     78.30
Average:        all      0.14      0.00     15.34     19.45      0.00     65.07
```

图 **4.1.7　sar** 的运行实例（**I/O** 密集型服务器）

“%iowait”是 I/O 等待率。load average 较高且该值较高时，可以判断是由于 I/O 瓶颈所造成的负载状况。

查明了具体是 I/O 的原因还是 CPU 的原因后，就可以通过其他指标来进行更加详细的调查，例如参考内存使用率和 SWAP 交换区的发生情

况等。

像这样，在辨别瓶颈时，从 load average 等总括性的数据着手，参考 CPU 使用率和 I/O 等待时间等具体的数字，从而自顶向下地快速排查各进程状态是非常有效的策略。而具体以怎样的顺序执行这些方针呢？当然需要结合当下内核的行为，以及此类负载数值的程序逻辑上的计算方法来决定。还是那句话，要想查看负载，必须清楚内核运作的情况。

4.1.7　多核 CPU 与 CPU 使用率

近来的基于 x86 的 CPU 正在向多核心（Multi-Core）结构发展。在多核 CPU 中，即便只有单个物理的 CPU，在系统看来也好像安装了多个 CPU 一样。Linux 内核中，CPU 使用率统计了各个 CPU 核心的详情。下面我们就来确认一下。

利用 `sar` 工具的 `-P` 选项。图 4.1.8 是安装了四核 CPU 的（Quad-Core CPU）服务器中`sar` 的结果。

```
% sar -P ALL | head -13
Linux 2.6.19.2-103.hatena.centos5 (jubuichi.hatena.ne.jp)      02/08/08

00:00:01        CPU     %user     %nice   %system   %iowait    %steal     %idle
00:10:01        all     59.84      0.00      1.54      0.00      0.00     38.62
00:10:01          0     68.10      0.00      3.71      0.00      0.00     28.19
00:10:01          1     52.82      0.00      0.81      0.00      0.00     46.37
00:10:01          2     53.52      0.00      0.76      0.00      0.00     45.72
00:10:01          3     64.94      0.00      0.88      0.00      0.00     34.18
00:20:02        all     48.72      0.00      1.48      0.00      0.00     49.80
00:20:02          0     62.81      0.00      3.59      0.01      0.00     33.59
00:20:02          1     39.11      0.00      0.81      0.01      0.00     60.07
00:20:02          2     38.17      0.00      0.71      0.00      0.00     61.12
00:20:02          3     54.79      0.00      0.82      0.00      0.00     44.39
```

图 **4.1.8　sar -P** 的运行实例（计算密集型服务器，安装了多核心的 **CPU**）

各 CPU（各核心）都附带有 CPU ID 号码，在输出的 CPU 信息中可以通过此号码进行确认，以得到每个 CPU 的使用率情况。

这是计算密集型服务器的情况，下面再来看看在 I/O 密集型服务器中的结果。首先不使用-P 选项，只查看统计结果（图 4.1.9）。

```
% sar | head
Linux 2.6.18-8.1.8.el5 (takehira.hatena.ne.jp)  02/08/08

00:00:01         CPU     %user     %nice   %system   %iowait    %steal     %id
00:10:01         all      0.14      0.00     17.22     22.88      0.00     59.
00:20:01         all      0.15      0.00     16.00     22.84      0.00     61.
00:30:01         all      0.16      0.00     19.66     18.99      0.00     61.
```

图 **4.1.9**　**sar** 的运行实例（**I/O** 密集型的服务器，安装了多核心的 **CPU**）

可以看出 I/O 等待（%iowait 栏）平均为 20% 左右。该服务器使用了双核心的 CPU（Dual Core CPU），可以通过`sar –P` 查看各 CPU 及整个 CPU 的 I/O 等待情况（图 4.1.10）。

```
% sar -P ALL | head
Linux 2.6.18-8.1.8.el5 (takehira.hatena.ne.jp)  02/08/08

00:00:01         CPU     %user     %nice   %system   %iowait     %steal    %id
00:10:01         all      0.14      0.00     17.22     22.88       0.00    59.
00:10:01           0      0.28      0.00     34.04     45.58       0.00    20.
00:10:01           1      0.00      0.00      0.40      0.18       0.00    99.
00:20:01         all      0.15      0.00     16.00     22.84       0.00    61.
00:20:01           0      0.30      0.00     31.61     45.58       0.00    22.
00:20:01           1      0.00      0.00      0.38      0.11       0.00    99.
```

图 **4.1.10**　**sar -P** 的运行实例（**I/O** 密集型的服务器，安装了多核心的 **CPU**）

结果有点儿意外。I/O 等待大体上只是在 CPU 0 发生着，CPU 1 则几乎没有工作。

在安装了多核心的 CPU 且只存在一个磁盘时，单位核心的 CPU 负载可

以分流到其他的 CPU 核心，但 I/O 负载却不能进行分流，因此 `sar` 的输出结果就会出现偏差。虽然 I/O 等待平均为 20% 左右，并不是太高，但可能会造成整个 CPU 的 I/O 等待数值产生明显偏差。因此在安装多核处理器的环境中，根据情况还需要查看 CPU 中每个核心的使用率情况。

4.1.8 如何计算 CPU 的使用率

与 load average 相同，了解 CPU 使用率的具体计算方法，对分析 `sar` 和 `top` 命令的输出结果是很有帮助的。此外也能更清楚地了解所输出的多核 CPU 占用率各项数值的意义。

得出 CPU 使用率的步骤与 load average 类似，即通过计时器中断在内核中进行 [3] 。

> [3] 单核心和多核心中使用的中断信号存在差异，但两者所使用的信号都是硬件以一定的周期发出的。

load average 是计算 CPU 相关运行队列中正在保持的进程描述符的数量。load average 的数值存放在内核的全局数组中。

而 CPU 使用率的运算则有些许不同。CPU 使用率的计算结果不是放置到全局数组中，而是在为每个 CPU 所准备的专门区域中进行存放 [4] 。正因为为每个 CPU 划分了专门的区域存放数据，因此在`sar` 中才可以获得每个 CPU 的具体信息。

> [4] 具体是在内核中的 cpu_usage_stat 结构体中实现的。

为了完成进程的切换工作，内核会在各进程生成之后记录进程所使用的 CPU 时间，这通常被称为“进程记账”（Process Accounting）。根据该进程记账获得的记录，调度器可以相应地降低 CPU 占用时间过长的进程的优先级，或者在该进程进行一定的计算后，将 CPU 让渡给其他的进程。

通过统计各进程的 CPU 使用时间，可以了解 CPU 被占用的情况。将其换算为单位时间的计算结果，就可以得出 CPU 使用率。

这里有如下两处关键点：

- **load average** 是系统全局的计算结果
- **CPU** 使用率和 **I/O** 等待时间是根据类型及具体使用的 **CPU** 来保存的计算结果

据此就可以弄清楚两个指标的差异：

- **load average** 是整个系统的负载的指标，不能进行详细的分析
- **CPU** 使用率和 **I/O** 等待时间既可以作为整体的统计报告来看，也可以作为具体的指标来看

4.1.9　进程记账的内核编码

在了解了进程记账的概要之后，为了能更确切地理解，还需要查看进程记账的实际编码。首先，查看 include/linux/kernel_stat.h 中定义的 **cpu_usage_stat** 结构体 和 **kernel_stat** 结构体 。

```
struct cpu_usage_stat

 {
    cputime64_t user;
    cputime64_t nice;
    cputime64_t system;
    cputime64_t softirq;
    cputime64_t irq;
    cputime64_t idle;
    cputime64_t iowait;
    cputime64_t steal;
};

struct kernel_stat

 {
    struct cpu_usage_stat   cpustat;
    unsigned int irqs[NR_IRQS];
};
DECLARE_PER_CPU(struct kernel_stat, kstat);
```

该结构中记录并保存了计算得出的 CPU 使用时间等数据。

请看 cpu_usage_stat 结构体，可以发现，`sar` 所输出的项目，如 user、

system、iowait 等成员都可以在该结构体中确认。kernel_stat 结构体中包含了 cpu_usage_stat 结构体，另外在 kernel_stat 结构体中还使用了 DECLARE_PER_CPU() 宏声明了每个 CPU。

进程记账的实际处理是在 kernel/timer.c 的 **update_process_times()** 函数中定义的。每当计时器中断时，该函数都会被调用。在 update_process_times() 内，可以判断从之前的进程记账处理开始到现在这段时间内当前进程的行为，并更新统计信息。

```
void update_process_times

(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id();

    /* Note: this timer irq context must be accounted for as well. */
    if (user_tick)
        account_user_time(p, jiffies_to_cputime(1));
    else
        account_system_time(p, HARDIRQ_OFFSET, jiffies_to_cputime(1));

    <省略>
}
```

首先，通过 current 宏来获取当前进程的进程描述符。然后根据 **user_tick** 的数值来判断分支。user_tick 会判断最近的时间具体是用户时间还是系统时间。

如果是用户时间，则调用 **account_user_time()** 函数，否则则调用 **account_system_time()** 函数。以下来看看 account_user_time() 的实现机制。

```
void account_user_time

(struct task_struct *p, cputime_t cputime)
{
    struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
    cputime64_t tmp;

    p->utime = cputime_add(p->utime, cputime);
```

```
    /* Add user time to cpustat. */
    tmp = cputime_to_cputime64(cputime);
    if (TASK_NICE(p) > 0)
        cpustat->nice = cputime64_add(cpustat->nice, tmp);
    else
        cpustat->user = cputime64_add(cpustat->user, tmp);
}
```

通过参数传递的“p”是当前进程的进程描述符。

首先，获取运行中的 CPU 所用的 cpustat 结构体。接着，使用 cputime_add 宏更新当前进程的 utime 成员。这样一来，该进程在用户模式所消耗的 CPU 时间的数值就完成了更新。然后，针对 cpustat 结构体的 nice 或 user 数值，使用 cputime64_add 宏以同样的方式算出经过时间。

另外，account_system_time() 是如何实现的呢？

```
void account_system_time(struct task_struct *p, int hardirq_offset, cputime
{
    struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
    struct rq *rq = this_rq();
    cputime64_t tmp;

    p->stime = cputime_add(p->stime, cputime);

    /* Add system time to cpustat. */
    tmp = cputime_to_cputime64(cputime);
    if (hardirq_count() - hardirq_offset)
        cpustat->irq = cputime64_add(cpustat->irq, tmp);
    else if (softirq_count())
        cpustat->softirq = cputime64_add(cpustat->softirq, tmp);
    else if (p != rq->idle)
        cpustat->system = cputime64_add(cpustat->system, tmp);
    else if (atomic_read(&rq->nr_iowait) > 0)
        cpustat->iowait = cputime64_add(cpustat->iowait, tmp);
    else
        cpustat->idle = cputime64_add(cpustat->idle, tmp);
    /* Account for system time used */
    acct_update_integrals(p);
}
```

虽然这里也以同样的方式进行了处理，但是之前的 update_process_times() 函数中所描述的“如果用户模式不工作则系统模式工作”只是一个比较粗略的条件分支。实际上当用户模式不工作时，还存在什么都没做的 idle 状态、系统模式计算时间、I/O 等待时间等情况。若要进行上述这些判断，需要深入了解 cpu_usage_stat 结构。

*　　*　　*

虽然稍微有些复杂，不过我们还是了解了 CPU 使用率统计的更新机制。这样一来，我们便能清楚地掌握`sar` 及`top` 命令中所罗列的各个指标具体是表示什么的了。

4.1.10　线程和进程

虽然稍微有些离题，但还是有必要稍微了解一下进程和线程。

通常情况下，线程是比进程还要小的运行单位。在进程中通常能够使多个线程同时运作，这就叫做多线程。若要在一个程序中同时并行多个处理，可以采用以下方式 [5]：

> [5] 也有在单位线程中通过事件驱动（Event Driven）进行处理的方法。

- 通过生成多个进程确保多个环境的运行（→ 多进程）
- 通过生成多个线程确保多个环境的运行（→ 多进程）

MySQL 是利用多线程来处理客户端的请求。Apache 中若将 MPM 选项设为“prefork”，就可以实现多进程；若选择“worker”，则可以实现多进程 + 多线程的运作。

多进程（图 4.1.11）和多线程（图 4.1.12）存在根本性的不同：前者拥有属于自己独立的内存空间；后者则是共享的内存空间。因此后者具有更经济的内存消耗，而且在进程切换时不发生内存空间置换，运算成本相对较低。需要大量运行环境的程序，采用多线程比较有利。

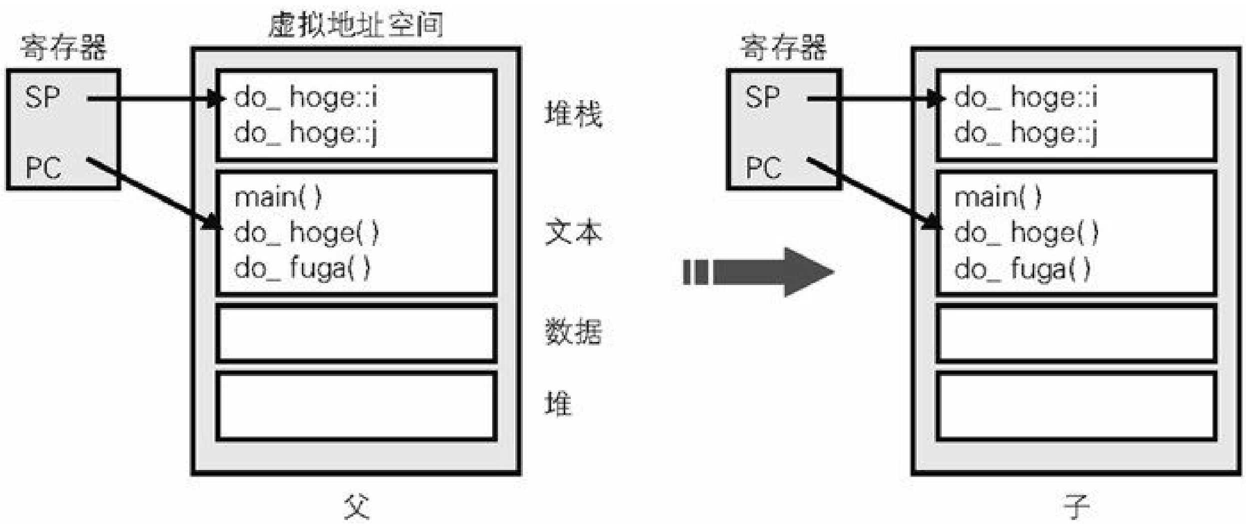


图 **4.1.11**　多进程（内存空间不同。复制）※

※ SP：栈指针； PC：程序计数器。

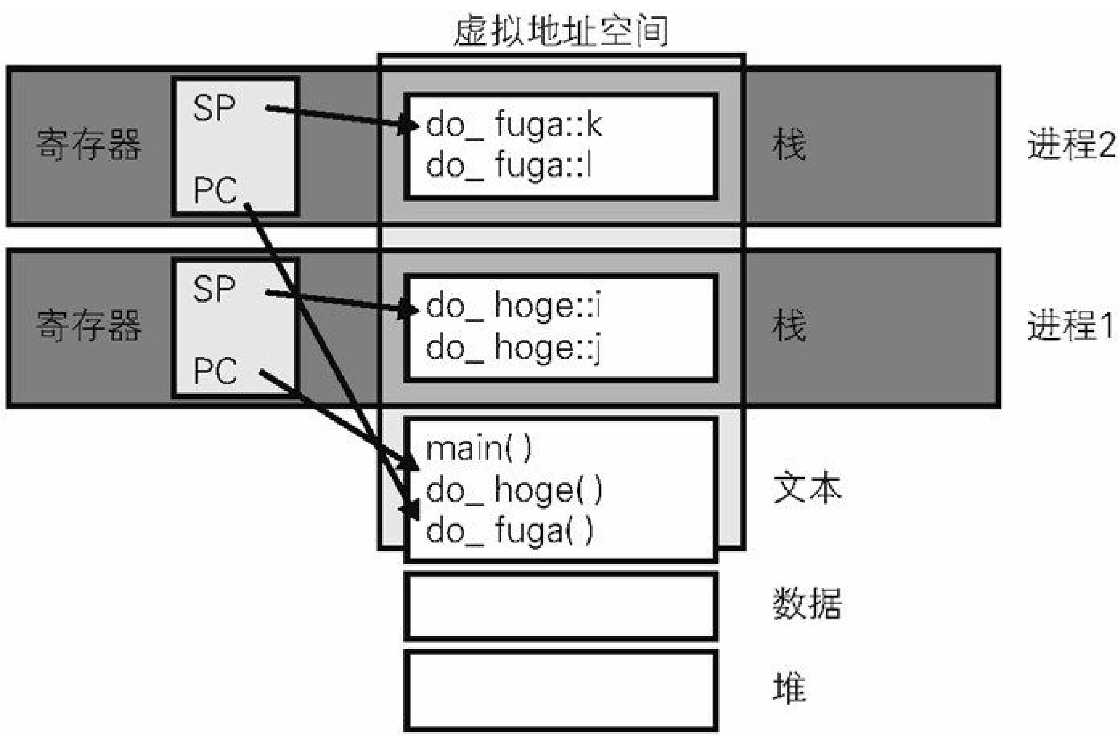


图 **4.1.12**　线程（内存空间相同）

内核中的进程和线程

但是以上只是从用户角度来看进程和线程的差异。

而在内核内部，进程和线程的处理方式则大致相同。针对一个线程分配一个进程描述符，进程和线程用完全一样的逻辑调度。因此，在运作多线程应用程序时，负载计算的方式也不会发生什么改变。

顺带一提，线程在内核中有时被称为 **LWP=Light Weight Process**，即轻量进程。

ps 和线程

对内核来说，进程和线程基本相同。但是从用户的角度来看，线程则是在进程中运行的运行环境。总之，线程是比进程更小的概念，进程包含线程。在通过`ps` 浏览多线程的全部线程时，需要相关的选项。

例如在查看 mysqld 的进程时，只能看到如图 4.1.13 所示的两个进程。这里如图 4.1.14 那样给`ps` 增加`-L` 选项。

```
% ps -elf | egrep (CMD|mysql)
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY
TIME CMD
4 S root      3297     1  0  81   0 - 13260 wait   Jan25 ?
00:00:00 /bin/sh /usr/bin/mysqld_safe
4 S mysql     3329  3297 99  75   0 - 100738 stext Jan25 ?
19-05:11:32 /usr/libexec/mysqld
```

图 **4.1.13**　**ps** 的运行实例

```
% ps -elf -L | egrep (CMD|mysql) | head
F S UID       PID

  PPID   LWP

  C NLWP

 PRI  NI ADDR SZ WCHAN  STIME TTY
TIME CMD
4 S root     3297     1  3297  0    1  81   0 - 13260 wait    Jan25 ?
00:00:00 /bin/sh /usr/bin/mysqld_safe
4 S mysql    3329
```

```
  3297  3329  0   37

  75   0 - 101251 -      Jan25 ?
00:11:23 /usr/libexec/mysqld
1 S mysql    3329

  3297  3332  0   37

  75   0 - 101251 -      Jan25 ?
00:03:44 /usr/libexec/mysqld
1 S mysql    3329

  3297  3333  0   37

  75   0 - 101251 -      Jan25 ?
00:03:44 /usr/libexec/mysqld
1 S mysql    3329

  3297  3334  0   37

  75   0 - 101251 -      Jan25 ?
01:00:09 /usr/libexec/mysqld
1 S mysql    3329

  3297  3335  0   37

  80   0 - 101251 -      Jan25 ?
00:00:00 /usr/libexec/mysqld
<以下省略>
```

图 **4.1.14**　使用 **-L** 选项显示单位进程的所有线程

这样一来就多出了几行输出的内容，在此多出的部分就是线程。请注意Header 的“PID”和“LWP”列。PID 是进程 ID，但是 mysqld 的进程 ID 完全相同。另一方面，LWP 是线程 ID。从进程 ID 相同但线程 ID 不同可以得出，这些线程是在同一进程内被建立的多个线程。

“NLWP”（Number of LWPs）是线程个数。在此可以看出，mysqld_safe 只有一个线程，也就是其自身，而 mysqld 线程却被生成了 37 个。

LinuxThreads 和 **NPTL**

由于历史原因，Linux 中有多个实现多线程的机制。虽然目前统一为了“NPTL”（Native POSIX Thread Library），但是稍旧的发行版还有可

能采用的是“LinuxThreads”实现机制。

LinuxThreads 和 NPTL 几乎没有什么区别，但是在使用 `ps` 查看时，LinuxThreads 的显示和进程几乎相同。在 NPTL 中不使用 `-L` 选项就不能确认线程，而使用 LinuxThreads 就算不添加 `-L` 选项也能确认线程，请注意不要混淆。

4.1.11 ps、sar、vmstat 的使用方法

下面我们回到正题。在此之前我们了解了：

- 负载监控的基本策略
- **load average** 的计算过程
- **CPU** 使用率的计算过程

理解了上面的内容，就可以明白各命令工具输出的各项指标。基于上文的知识，下文将深入介绍`ps`、`sar`、`vmstat` 等常用工具的使用方式。

ps ……输出进程信息

ps（Report Process Status）是输出进程信息的软件。是从用户空间调用内核中所保持的进程描述符中存储信息的工具。

让我们来看一下`ps auxw` 命令输出的信息（图 4.1.15），下面是其中主要的几列。

- **%CPU：** 运行 **ps** 命令时该进程的 **CPU** 使用率
- **%MEM：** 以百分比表示进程的物理内存消耗程度
- **VSZ、RSS：** 分别是该进程所确保的虚拟内存区域的大小及物理内存区域的大小（详情见后）
- **STAT：** 像之前介绍的那样，该项目显示了进程的状态。是非常重要的项目
- **TIME：** 表示 **CPU** 占用时间的项目（详情见后）

```
% ps auxw
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   1944   656 ?        Ss   Feb05   0:00 init [2
root         2  0.0  0.0      0     0 ?        S<   Feb05   0:00 [kthrea
root         3  0.0  0.0      0     0 ?        SN   Feb05   0:00 [ksofti
root         4  0.0  0.0      0     0 ?        S<   Feb05   0:00 [events
root         5  0.0  0.0      0     0 ?        S<   Feb05   0:00 [khelpe
root        17  0.0  0.0      0     0 ?        S<   Feb05   0:00 [kblock
root        18  0.0  0.0      0     0 ?        S<   Feb05   0:00 [kserio
root        34  0.0  0.0      0     0 ?        S    Feb05   0:00 [pdflus
root        35  0.0  0.0      0     0 ?        S    Feb05   0:07 [pdflus
root        36  0.0  0.0      0     0 ?        S<   Feb05   0:01 [kswapd
root        37  0.0  0.0      0     0 ?        S<   Feb05   0:00 [aio/0]
root       786  0.0  0.0      0     0 ?        S<   Feb05   0:03 [kjourn
root       982  0.0  0.0      0     0 ?        S<   Feb05   0:10 [kjourn
root       983  0.0  0.0      0     0 ?        S<   Feb05   0:01 [kjourn
root      1218  0.0  0.0   1628   616 ?        Ss   Feb05   0:00 /sbin/s
root      1224  0.0  0.0   1576   380 ?        Ss   Feb05   0:00 /sbin/k
```

图 **4.1.15**　确认 **ps auxw** 的输出

VSZ 与 **RSS ……**虚拟内存和物理内存的指标

VSZ（Virtual Set Size）是为了支持进程的正常运作所设置的虚拟内存区域的大小，RSS（Resident Set Size）是物理内存区域的大小，但这里为什么会出现两个内存的指标呢？

不仅是 Linux，多任务操作系统的一个重要功能就是存在虚拟内存的结构。所谓“虚拟内存”（Virtual Memory），是指当程序需要使用内存时，并不是直接让物理内存介入处理，而是让物理内存通过软件抽象层，实现被称为“页面文件”（Paging，也称虚拟内存）的虚拟内存结构，系统将对该虚拟内存区域进行相应的管理。

一些进程需要确保足量的内存空间才能正常工作，但由于多任务系统保护的关系，用户进程无法直接访问硬件，于是便暂时将处理中断，并委托内核来确保内存。

虽然内核必须确保对进程内存的分配，但这里并非交付真实的物理内存区域的地址，而是交付虚拟内存的地址。进程把从内核返回的虚拟内存

的地址当成真实地址，然后重新开始处理。

这里需要留意的是，内核为进程返回的虚拟内存地址，事实上此时还未与物理内存建立起实质的联系，也就是说可以认为在硬件上还没有划出实际的内存区域。直到进程对通过内核得到的虚拟内存区域进行写入时，对物理内存区域的关联操作才开始进行（图 4.1.16）。可以说内核用虚拟内存这个中间的抽象层（善意地）欺骗了进程。

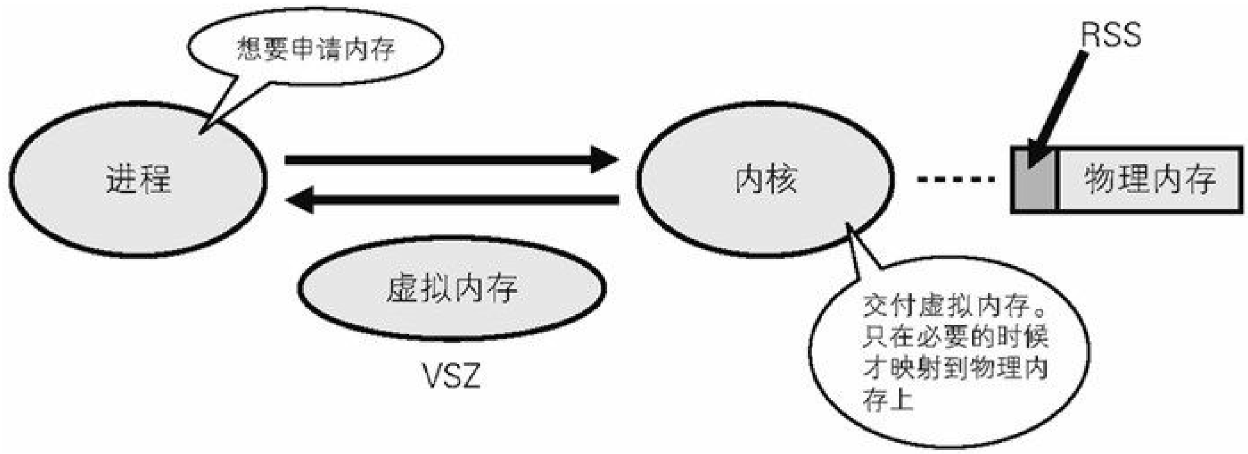


图 **4.1.16**　虚拟内存

使用虚拟内存结构能获得很大的便利，这也是支撑多任务操作系统的一项重要功能。以下列举几点：

- 通过使用虚拟内存可以欺骗进程，造成可以处理超出物理内存容量的更大内存空间的假象
- 使物理存储器上零乱的内存空间看起来像是一个连续的内存空间
- 使进程认为每个进程都拥有独立的内存空间
- 当物理内存不足时，长时间没被使用的虚拟内存空间将解除与物理内存空间的映射，被解除映射的数据将二次存放在存储装置（磁盘等）上，当需要这些数据时会进行还原操作，这通常被称为**“**交换**”**（**SWAP**）
- 在不同的两个进程上，所使用的虚拟内存空间也不同，通过将这两个虚拟内存空间映射到同一个物理内存空间，就可以在两个进程间共享内存。**IPC**[6] 共享内存等就是通过这种方式实现的

> [6] Inter Process Communication 的缩写，即进程之间的通信（功能）。

VSZ 及 RSS 分别会输出虚拟内存空间及物理内存空间的大小。当频繁访问交换区时，通常可以判断出此时的物理内存空间已经不足以使用，因此需要时长留意 RSS 的大小，针对占用 RSS 较大的进程查找问题根源以及时解决，或者扩充物理内存的容量以解决潜在的负载问题。

TIME ……CPU 占用时间

TIME 是表示时间的指标，在这里指的是进程实际所占用的 CPU 时长。请注意这里不是生成进程后经过的时长。

进程实际使用的 CPU 时间指的是什么呢？相信你已经注意到了！通过查看之前进程记账中的处理详情，就可以看到进程描述符中记录的 CPU 使用时间。例如，当系统 CPU 负载较高时，通过查看 `ps` 的 TIME 项，就可以分清具体是哪个进程 CPU 占用了较多资源。

通过 **ps** 命令查看 **Blocking** 和 **Busy Loop** 的差异

这里为了加深对 CPU 时间的理解，我们试着做一个实验。首先尝试运行两个无限循环的 Ruby 脚本，然后在 `ps` 中确认该进程的行为。第一个脚本是如代码清单 4.1.1 所示的只执行加法的脚本 (busy_loop.rb)。运行一会儿该脚本后再看看`ps` 输出的结果（图 4.1.17）[7] 。

> [7] 选项没有使用 BSD 形式的 auxw，而是使用了 SysV 形式的 -fl，虽然输出与先前介绍的稍有不同，但是可以看出信息并没有很大差异。

代码清单 **4.1.1　busy_loop.rb**

```
#!/usr/bin/env ruby

i = 0
while true
  i += 1
end
```

```
% ps -fl -C ruby
F S

 UID        PID  C        TIME

 CMD
0 R

 naoya    10640 69    00:00:23

 ruby busy_loop.rb
```

图 **4.1.17**　**ps** 的运行实例（**busy_loop.rb**，省略了部分列）

这里希望大家注意的是表示状态的“S”列和“TIME”列。代码清单 4.1.1 的脚本只在 CPU 上进行无限循环的加法运算，没有进入事件等待的处理，因而在“S”列中，可以看出该进程恒常显示为“R”，即“TASK_RUNNING”状态。因为 CPU 一直被消耗，所以 TIME 即 CPU 的占用时间会随着时间的流逝而增加。此类无限循环的 CPU 计算处理被称为 Busy Loop。

另一方面，代码清单 4.1.2 进行了什么样的行为呢？其实该脚本就是读取用户在键盘输入的内容的脚本（blocking.rb）。`ps` 的结果如图 4.1.18 所示。

代码清单 **4.1.2**　**blocking.rb**

```
#!/usr/bin/env ruby

while true
  puts gets
end
```

```
% ps -fl -C ruby
```

```
F S UID        PID  C        TIME CMD
0 S naoya    10753  0    00:00:00 ruby blocking.rb
```

图 **4.1.18　ps** 的运行实例（**blocking.rb**，省略了部分列）

由于等待键盘输入，该进程被 Block，因此状态是“S”，也就是“TASK_INTERRUPTIBLE”。还有，该进程只要变为待机状态，就不占用 CPU 时间。因此即使等待再长时间，TIME 的数值也不可能增加。

同样是无限循环，Busy Loop 的脚本和被 Block 的脚本的运作方式却存在很大差异，这通过`ps` 命令所呈现的项目也可以看出。若能清楚地了解进程的状态变迁和 CPU 占用时间的计算机制，就能胸有成竹地阅读 `ps` 中各个列的项目了。

sar ……查看系统报告的各项指标

可以查看系统报告的各种指标的工具有很多，其中通用又方便的是 **sar**（System Activity Reporter）。

`sar` 是存在于 sysstat 软件包中的指令，有两个用法：

- 追溯访问过去的统计数据（默认）
- 周期性地确认当前数据

在`sar` 中，运行了 sadc 的后台程序。若安装了 sysstat 软件包，sadc 就能自动从内核收集报告进行存储。如前所述，如果不附加任何选项直接运行 `sar` 指令的话，就可以参考 sadc 所收集的过去的 CPU 使用率的统计数据。

默认显示从最近的 0:00 开始的数据。想查阅昨天以前的报告时，可以像图 4.1.19 那样用`-f` 选项指定 /var/log/sa 目录下保存的日志文件。

```
% sar -f /var/log/sa/sa04 | head
Linux 2.6.19.2-103.hatena.centos5 (goka.hatena.ne.jp)   02/04/08

00:00:01     CPU   %user    %nice   %system   %iowait    %steal    %idle
```

```
00:10:01      all     3.21       0.00       2.51       2.16       0.00      92.12
00:20:01      all     3.10       0.00       2.48       2.04       0.00      92.38
00:30:01      all     3.01       0.00       2.34       1.94       0.00      92.71
00:40:02      all     2.92       0.00       2.29       1.95       0.00      92.84
```

图 **4.1.19**　**sar -f** 的运行实例

查阅过去的数据的功能非常重要。例如在发生故障等情况下，为了探寻故障的原因，故障发生前后的数据就很有价值。另外还能通过 sar 的数据来比较更换程序前后的性能变化，以确认此次程序部署是否存在价值。

如果不查看过去的数据，而是查看当前数据，可以使用类似sar 1 1000 这样的参数格式。“1 1000 ”是“每隔 1 秒采样一次，连续采样 1000 次”的意思。

如图 4.1.20，可以查阅每秒钟的 CPU 使用率。如果要确认现在系统正发生着什么，多数情况下使用sar 即可。

```
% sar 1 3
Linux 2.6.19.2-103.hatena.centos5 (goka.hatena.ne.jp)     02/08/08

16:13:30      CPU   %user   %nice   %system   %iowait   %steal   %idle
16:13:31      all    2.04    0.00      3.56      3.82     0.00   90.59
16:13:32      all    2.27    0.00      2.02      1.26     0.00   94.44
16:13:33      all    2.28    0.00      2.03      1.52     0.00   94.16
Average:      all    2.20    0.00      2.54      2.20     0.00   93.07
```

图 **4.1.20**　通过 **sar** 命令来查看当前数据

可以为sar 命令指定选项，以查看 CPU 使用率以外的各数值。虽然能够通过不同的选项来查阅大量的报告，但这里只集中介绍几种比较常用的选项的使用方式。另外，正如上文所介绍的那样，用-P 选项可以查阅每个 CPU 的数据。

sar -u ……查看 **CPU** 的使用率

查看默认情况下的 CPU 使用率等信息可以使用 `sar -u` 命令（图 4.1.21）。各列指标分别为

- **user：** 在用户模式中，**CPU** 被消耗的时间比例
- **nice：** 通过 **nice** 变更了调度优先级的进程，在用户模式下所消耗的 **CPU** 时间的比例
- **system：** 在系统模式中，**CPU** 被消耗的时间比例
- **iowait：** 磁盘 **I/O** 等待的 **Idol** 状态消耗的 **CPU** 时间比例
- **steal：** 利用 **Xen** 等系统虚拟化技术时，等待其他虚拟 **CPU** 计算所占用的时间比例
- **idle：** **CPU** 没有等待磁盘 **I/O** 的空闲时间所占用的时间比例

```
% sar -u 1 3
Linux 2.6.19.2-103.hatena.centos5 (koesaka.hatena.ne.jp)   02/08/08

16:19:14     CPU   %user   %nice   %system   %iowait   %steal   %idle
16:19:15     all   14.89    0.00      1.74      0.00     0.00   83.37
16:19:16     all   26.37    0.00      1.49      0.00     0.00   72.14
16:19:17     all   17.00    0.00      1.50      0.00     0.00   81.50
Average:     all   19.42    0.00      1.58      0.00     0.00   79.00
```

图 **4.1.21** **sar -u** 的运行实例

像迄今为止所看到的那样，在考虑负载分流时，user/system/iowait/idle 等的数值将被列为重要的参考指标。

sar -q ……查看 **load average**

指定`-q` 参数，可以查看运行队列中存在的进程数、系统中进程的大小及 load average 等（图 4.1.22）。该报告的数值会随着时间发生变化，比其他命令更方便。

```
% sar -q 1 3
Linux 2.6.19.2-103.hatena.centos5 (koesaka.hatena.ne.jp)  02/08/08

16:15:19      runq-sz  plist-sz   ldavg-1   ldavg-5  ldavg-15
16:15:20            0       123      0.62      0.72      0.81
16:15:21            0       123      0.62      0.72      0.81
16:15:22            2       122      0.62      0.72      0.81
Average:            1       123      0.62      0.72      0.81
```

图 **4.1.22**　**sar -q** 的运行实例

sar -r ……查看内存的使用状况

指定 `-r` 参数，可以浏览物理内存的使用状况。图 4.1.23 是在安装有 4GB 的物理内存的服务器上执行 `sar -r` 命令的结果。各列的 kbmemfree 和 kbmemused 中的"kb"是 Kilobyte 的缩写。其中几个主要项目的意义如下。

- **kbmemfree**： 可用的物理内存容量
- **kbmemuserd**： 使用中的物理内存容量
- **memused**： 物理内存的使用率
- **kbbuffers**： 被作为内核上的缓冲区使用的物理内存容量
- **kbcached**： 内核中被作为缓存使用的物理内存容量
- **kbswpfree**： 可用的交换区容量
- **kbswpued**： 使用中的交换区容量

```
% sar -r | head
Linux 2.6.19.2-103.hatena.centos5 (koesaka.hatena.ne.jp)       02/08/

00:00:01   kbmemfree kbmemused  %memused kbbuffers  kbcached kbswpfree
00:10:01      522724   3454812     86.86    114516   2236880   2048204
00:20:01      534972   3442564     86.55    114932   2225880   2048204
00:30:01      437964   3539572     88.99    115348   2238952   2048204
```

```
00:40:01      491184   3486352     87.65    115768   2251440   2048204
00:50:01      491208   3486328     87.65    116160   2263248   2048204
01:00:01      457364   3520172     88.50    116524   2274732   2048204
01:10:01      453172   3524364     88.61    116904   2281576   2048204
```

图 **4.1.23**　**sar -r** 的运行实例（省略部分列）

使用 `sar -r`，可以随着时间的推移掌握内存的使用情况，即了解内存具体被用在了什么地方，被使用了多大比例等。若与后述的 `sar -W` 配合使用，还可以在频繁访问交换区时，了解该时间段内内存的使用情况。

减轻 **I/O** 负载及页面缓存

在图 4.1.23 中，“%memused”显示为了 90% 左右的数字，可用容量仅为 500MB（Megabyte）左右。随着时间的推移，可用容量 kbmemfree 将越来越少，可以判断出未来内存的使用可能会越来越紧张。但是这里我们忘记了 Linux 中“页面缓存”（Page Cache）的存在。

Linux 从磁盘读出一次数据后，会尽可能地在内存中进行缓存，以加速下次的磁盘读取 (Disk Read) 速度。像这样，从内存中读取的数据的缓存就被称为页面缓存。

Linux 是将内存切分为 4KB（Kilobyte）的块进行管理的，该 4KB 的块就被称为“页面”(Page)。页面缓存，顾名思义，是通过缓存页面进行工作的。也就是说，从磁盘读取数据时就建立相应的页面缓存，这样在下次读出数据时，直接将数据从页面缓存转送到用户空间即可。

请记住 Linux 平台中页面缓存的行为策略，即 Linux 将尽可能地利用更多的内存来转送页面缓存。也就是说：

- 无论在磁盘上读取什么数据，
- 只要该数据在页面缓存上不存在，
- 并且有空余的内存空间，

- 就在页面缓存中建立新的缓存（而不是替代旧的缓存）

如果当前没有足够的内存用来缓存，就丢弃旧的缓存以更换为新的缓存。当进程需要内存空间时，将优先释放页面缓存以分配内存。

在`sar -r`的结果中，随着时间的推移 kbmemfree 会不断减少，这是因为存在页面缓存的缘故，从页面缓存所分配的内存容量即 kbcached 的数值在不断增加这一点可以看出。

通过页面缓存减轻 **I/O** 负载的实施效果

页面缓存可以多大程度地减轻负载呢？如果数据被完全缓存在内存中，因为几乎所有请求都是在读取内存，因此可以预见读取速度将和程序直接读取内存的速度无异。

例如，将实际运行 MySQL 的数据库服务器的内存从 8GB 增加到 16GB，增加前后的`sar -P 0`的输出比较如图 4.1.24 所示。由于该数据库保存的数据不到 20GB，因此如果有 16GB 的内存，就可以缓存大部分的有效数据。

```
●内存为8GB时
13:40:01        CPU      %user     %nice    %system    %iowait

     %idle
13:50:01          0      20.57      0.00      15.61      23.90

     39.92
14:00:01          0      18.65      0.00      16.54      30.36

     34.45
14:10:01          0      19.50      0.00      15.26      20.51

     44.73
14:20:01          0      19.38      0.00      16.19      21.93

     42.50

●内存增加后
15:20:01        CPU      %user     %nice    %system    %iowait

     %idle
15:30:01          0      23.31      0.00      17.56       0.81
```

```
     58.32
15:40:01          0      22.43      0.00       16.60       0.86

     60.11
15:50:01          0      22.90      0.00       16.93       1.06

     59.11
16:00:01          0      23.54      0.00       18.37       1.02

     57.07
```

图 **4.1.24　sar -P 0** 的输出比较

增加内存的效果一目了然，原本高达 20% 的 I/O 等待（%iowait）几乎不存在了。

可见，特别是在 I/O 密集型服务器中，减轻 I/O 负载的有效的方法就是结合服务器处理的数据量安装相匹配的内存。

通过运行 `sar -r` 命令，可以判断内核确保了多少缓存。比较缓存的容量及实际应用程序处理的有效数据量，如果数据量较多，就需要考虑增加内存。通过将数据合理地进行缓存，就能将磁盘的访问频率降至最低。使用后述的`vmstat` 工具，就能确认实际的磁盘访问频率如何。

在不能增加内存时，可以考虑将数据分割到不同的主机上。顺利分割数据后，不仅能够降低磁盘 I/O 的负载，由于扩充了缓存中的数据容载量，因此还可以大幅度提高设备的吞吐量。

将所需的数据整个放到页面缓存上

如前所述，页面缓存就是缓存，在缓存失败时，数据自然要从硬盘上读取。由于系统启动后大部分数据都是未缓存的状态，因此大部分的读取请求都会被转送到硬盘而非缓存上。

在运行 MySQL 等的数据库服务器的环境中，当处理大规模的数据时，需要特别注意这一点。

例如，在重新启动服务器进行维护等时，之前在内存中被缓存的页面缓存将全部被清空。那么在没有建立相应的缓存的情况下，实际运行请求数很多的数据库服务器会发生什么呢？可以想象，约莫所有的数据库访

问请求都会造成磁盘 I/O 的负载压力。在大规模的环境中，基于这个原因致使数据库被锁定，导致服务暂停的情况屡见不鲜。因此在生产环境中，在重启服务器等操作后，需要首先将所需的数据整个放到页面缓存上。

例如，当 I/O 密集型服务器的 I/O 负载较高，吞吐数据有困难时，是否对页面缓存进行优化，差异是很明显的。

图 4.1.25 中介绍了一个很典型的数据报表。这是在安装有 4GB 内存的 MySQL 服务器中，在系统启动后 20 分钟左右使用 `sar -r` 命令输出的结果。在系统启动后，运行了读取 MySQL 中所有数据文件的（只读）程序。

```
18:20:01     kbmemfree kbmemused   %memused

 kbbuffers   kbcached
18:30:01       3566992    157272       4.22

     11224      50136
18:40:01       3546264    178000       4.78

     12752      66548
18:50:01        112628   3611636      96.98

      4312    3499144
```

图 **4.1.25**　保持页面缓存的实例（省略部分列）

启动该程序前，内存使用率不到 5%，大约有 3.5GB 的可用内存。启动该程序读取数据文件后，内存使用率提高到了 96.98%。这是因为程序读取了相应的数据文件，此时已经将文件内容放到了页面缓存上。

sar -W ……查看交换区的吞吐状况

指定`-W`参数，可以确认交换区的吞吐状况（图 4.1.26）。“pswpin/s”是每秒系统换入的页面数，“pswpout”则与之相反，是每秒系统换出的页面数。发生频繁的交换时，服务器的吞吐量性能会大幅下降。当服务器的状态不理想时，可以利用 `sar -W` 检测是否是由于内存不足的原因使交换区发生了频繁的交换。

```
19:20:01      pswpin/s pswpout/s


19:30:01          0.00      0.00
19:40:01          0.00      0.00
19:50:37         44.01    811.27
Average:          0.39      7.21
```

图 **4.1.26　sar -W** 的运行实例

vmstat ……查看虚拟内存的相关信息

简单介绍一下 **vmstat** （Report Virtual Memory Statistics）的使用方法。vmstat 的“vm”是指 Virtual Memory（虚拟内存）。vmstat 是可以查看虚拟内存相关信息的工具。大多数指标也都可通过 sar 命令查看，但该命令的方便之处在于可以实时确认 CPU 使用率及实际的 I/O 等待时间。

vmstat 和sar 的使用方法很相似。vmstat 1 100 是“每隔 1 秒采样一次，连续采样 1000 次”的意思。

图 4.1.27 是 vmstat 的输出实例。各个项目的意义请参阅 man vmstat 的帮助文件。但通过之前的说明，从项目名称应该就能想象到大概的意义。

```
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu--
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id
 0  0      4  61692 342476 118464    0    0     3    16  105    41  1  1 98
 0  0      4  61692 342476 118464    0    0     0     0  101    12  0  0 100
 0  0      4  61692 342480 118464    0    0     0   192  101    18  0  0 100
 0  0      4  61692 342480 118464    0    0     0     0  101    10  0  0 100
```

图 **4.1.27　vmstat** 的输出实例

图 4.1.27 中所示的“bi”和“bo”的数值的意义分别为

- **bi**： 每秒从块设备接收到的块数，即读块设备（**blocks/s**）

- **bo**：每秒发送到块设备的块数，即写块设备（**blocks/s**）

所谓块设备（Block Device），坦白说就是二次存储装置，也就是磁盘。Linux 是把硬件的输入输出分成以下两类进行处理的。

- 字符设备（**Character Device**）：以字节为单位进行输入输出的硬件

- 块设备：简称为块，以一定大小的块为单位进行输入输出的硬件

磁盘就相当于这个块设备。通过 `vmstat` 命令，可以将此时磁盘的读（bi）写（bo）情况以块为单位显示出来。

通过 `top` 和 `sar` 命令，可以同时确认 CPU 使用率及 I/O 等待时间。但是从 I/O 等待的数字我们只能看出系统宏观的 I/O 等待时间，如果想知道具体哪里发生了 I/O 的读写，可以使用`vmstat` 命令来进行确认。

4.1.12　找到系统负载的症结并解决

明白了如何监控负载之后，接下来就要进入优化系统的课题了。虽然这么说，但是本书中并不会做过多讲解。提到“优化”这个词语，说不定有人会想这是要将软件性能提高两三倍呢。

但其实优化的真正工作是“找出系统瓶颈并加以解决”。首先需要了解到的是，想要突破硬件或软件原本的性能是怎样努力都无法达到的，我们所能做的就是“充分发挥硬 / 软件本来的性能，解决可能存在的问题”。

最近的系统和中间件，默认的设定状态就可以充分发挥性能。就像即使拓宽了不拥堵的高速公路车道，一辆汽车到达目的地的时间也不会缩短。如果默认的设定已经是最优，那么无论再怎么改变设定，大多数情况下也是没有效果的。

例如，逻辑最优的情况下，CPU 也需要 10 秒时间计算处理，无论怎样改动系统的设置也不可能缩短至 10 秒以下。这就是一个典型的不拥堵的高速公路的例子。

但若是程序的 I/O 性能存在问题，该程序本来 10 秒就可以结束，但结果花费了 100 秒才完成 I/O 的读写，这种情况下就可以优化 I/O 性能。

这个是拥堵的高速公路的例子。为了改善 I/O 性能，需要考虑如下问题：

- 能否通过增加内存确保足量的缓存空间来解决
- 原本数据量是否过多
- 是否需要变更应用程序方面的 **I/O** 算法

知道了问题的原因之后，根据经验很快就能找到适当的解决办法。而实践这一解决方法的过程，就是所谓的优化。

最后再强调一遍，为了最大限度地发挥硬件及系统的性能，需要掌握足够的经验，例如发生瓶颈时需要能清楚地判明具体是哪里出现了瓶颈等。本章中所说明的有关系统内部的实现机制和负载监控的方法，都是最基础的知识。

4.2　Apache 的优化

4.2.1　Web 服务器的优化

截止到目前都是在围绕系统进行讲述，下面我们将目光转向系统上运行的应用程序、Web 服务器。这里主要介绍广泛使用的 Web 服务器：Apache HTTP SERVER（Apache）[8]。

> [8] ***URL*** http://httpd.apache.org/

与系统的优化类似，Web 服务器的优化也并不是说能够让 Web 服务器的性能有两三倍的提高。这里的优化是指充分发挥服务器原本应有的性能。

4.2.2　Web 服务器遭遇瓶颈怎么办

其实，当 Web 服务器因超载而不能很好地返回响应时，问题基本上与 Web 服务器的配置并无太大关系。Web 服务器是相对比较稳定的软件，在系统上并不消耗过多的资源。

问题只是无响应的问题在 Web 服务器上表现得比较明显，而其实症结并不一定在 Web 服务器。就像生病出现了发烧现象一样，光是退烧，病是无法治愈的。

在这样的情况下，无论如何改动 Apache 的配置，只要其他地方还存在问题，那么再怎么调整 Apache 的配置也是没有意义的，请首先认识到这一点。正如在 4.1 节中所介绍的那样，我们不能单改变 Apache 的配置来监控系统的情况，查出问题的根源是最重要的。这里也需要做到“别臆断，请监控”。很多问题都可以使用目前为止了解到的 `ps` 、 `sar` 、 `vmstat` 等工具找出。

当硬件和系统充分发挥其性能时，是不太会出现负载超标的情况的，但是一旦发生负载超标，就需要仔细权衡 Web 服务器的各项配置。本文将从 Apache 的配置项目中，选取几个在大规模并发的生产环境中可能会影响性能的设置选项进行讲解。

4.2.3　Apache 的并发处理与 MPM

在讲解 Apache 的配置项目之前，先来回顾一下 Apache 的并发处理的结构。

不仅限于 Apache，面向非特定的多个客户端的公网服务器都需要能够并发处理来自多个客户端的请求。若不进行并发处理，在一个客户端连接服务器进行输入输出期间，其他客户端将无法连接到该服务器（图 4.2.1）。特别是以 Apache 为代表的 Web 服务器，如何在同一时间处理多个连接是决定性能的根本因素，并发处理的正确实现会对服务器的性能带来很大影响。

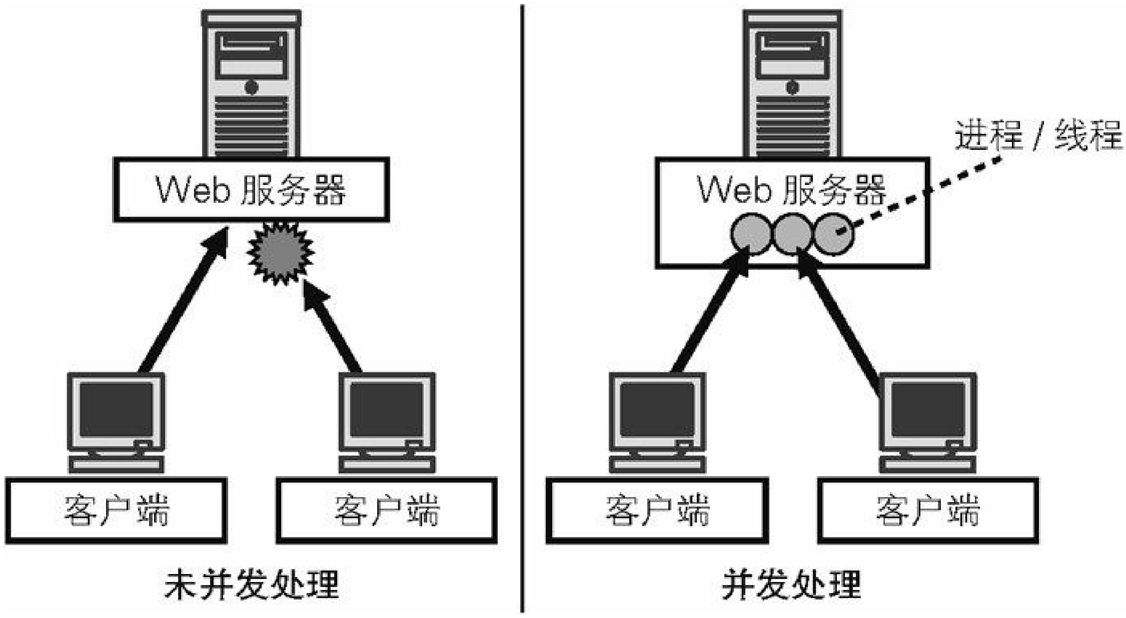


图 **4.2.1**　能 / 否并发处理

目前常用的有以下几种并发处理模式：

- 通过生成多个进程实现并发处理的多进程模式
- 不通过进程，而使用更轻量的运行单位——线程的多线程模式
- 通过监控输入输出事件，在事件发生时进行切换处理，即可使用单线程进行并发处理，这就是事件驱动的处理模式

这些模式各有优缺点，不能肯定哪个最好。目前也有综合了这些模式的应用。

Apache 通过模块化清楚地分离了内部的各种功能，进行并发处理的核心部分也单独成为了模块的结构。在这里，模块通常被称为 MPM（Multi Processing Module）。根据所选择的 MPM 的不同，用户可以决定使用不同的并发处理模式。在 Apache 2.2 中可以使用的 MPM 可通过以下链接进行确认。

URL http://httpd.apache.org/docs/2.2/zh-cn/mod/

UNIX 环境中最具代表性的 MPM 是以下两个（图 4.2.2）[9]：

[9] 此外还有在 worker 上融合事件模式的优势的“event MPM”，在 Apache 2.2 版本上该模块可以在实验的环境中使用，没有被过多地利用在生产中。

- **prefork**：提前生成（**Prefork**）多个进程以供客户端连接的多进程模式
- **worker**：多线程和多进程的混合型模式

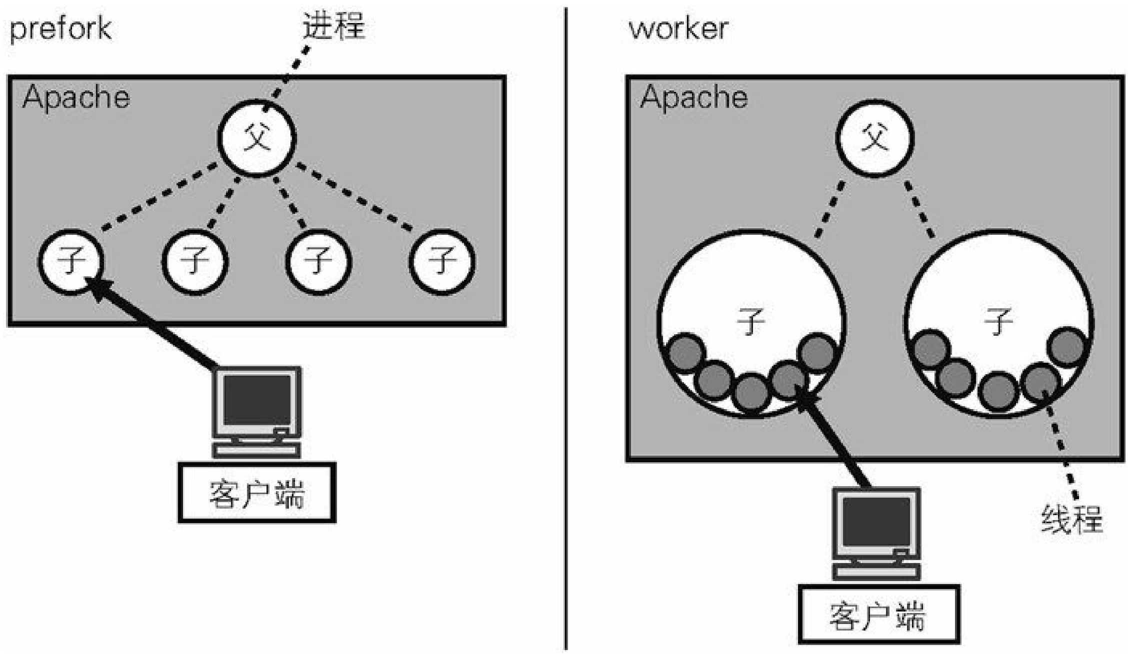


图 **4.2.2**　**UNIX** 环境中具有代表性的 **MPM**

由于是在编译时决定具体使用哪种 MPM 模式的，因此之后再决定使用其他 MPM 时基本上就需要重新编译 Apache。但在 Red Hat Enterprise Linux 及基于 Red Hat 的 Cent OS 的 Linux 上，同时安装了支持 prefork 和 worker 的两种 httpd。默认可以使用 prefork 混合模式，若要切换到 worker 模式，可以在 /etc/sysconfig/httpd 中进行如下设定：

```
HTTPD=/usr/sbin/httpd.worker
```

prefork 与 **worker**，进程与线程

prefork 是多进程模式，worker 是多线程和多进程的混合型模式。由于后者所占用的内存相对较小，因此在大规模并发环境中更适合使用 worker 模式。让我们再详细说明一下。

从编程模型看多进程 / 多线程的差异

Apache 不仅可以返回单一的 HTML 或图片等静态文件，还可以结合 mod_perl 和 mod_php 等模块被作为 AP 服务器来使用，或者如 2.1 节 中所介绍的那样结合 mod_proxy_balancer 模块被作为反向代理使用，可以看出模块的选择可以决定 Apache 的功能。Apache 2.2 的使用范围则更加广泛（当然也存在一些异议），如今 Apache 与其说是 Web 服务器，不如说是一个通用的网络服务平台。

一般来说，“多进程”和“多线程”之中，后者的编程模型往往更加复杂。这里我们再来重新确认一下图 4.1.11 及图 4.1.12。

- 多进程中基本上不在进程之间直接共享内存。内存空间是独立且安全的
- 在多线程中，多个线程共享内存空间，需要留意不能发生资源冲突。这也是多线程编程复杂的原因

因此，在第三方的 Apache 模块中，存在不能在多线程环境中正常工作的模块，以及不是基于多线程运行的模块，这些模块都是以使用 prefork 为前提的。

因此，我们可以对这两种模块做出如下定位。

- **prefork：** 该 **MPM** 具有更高的稳定性和向后兼容性
- **worker：** 该 **MPM** 的可扩展性更强

需要考虑使用第三方模块时使用 worker，或者在使用第三方模块时，根据该模块的规格合理选择 prefork 或 worker，这是一个应该遵循的准则 [10]。

> [10] 在 worker 中运行 mod_perl 时，Perl 将通过 ithreads 生成线程。由于 Perl 的线程多少有些特殊，在 prefork 的情况下可能存在一些规格的差异，因此有很多用户因为讨厌这一点而选择 prefork 模式。

从性能的观点来看多进程 / 多线程的差异

通常情况下，在多进程和多线程中，后者更轻量更快。主要原因有以下两点：

❶ 多进程使用多个独立的内存空间，而多线程只使用共享的内存空间，因此内存消耗量较少

❷ 因为多线程共享内存空间，所以线程切换的成本低于多进程

在 Apache 中，基于什么因素来决定选择哪种模式呢？

关于 ❶，从内存占用量方面来说确实使用多线程的 worker 更胜一筹。但事实上即使在使用多进程的情况下，由于内存空间在没有更新时是被父子进程所共享的（即写时复制技术，Copy-on-Write），因此性能上并没有显著差异。关于写时复制技术后面会有详细讲解。

❷ 其实是在讲上下文切换（Context Switch，也称文本切换）的成本差别。在多任务操作系统中，为实现并发处理，需要在短时间内切换 处理内容不同的进程 / 线程 [11]，此时的进程 / 线程的转换处理就被称为“上下文切换”。在进行上下文切换时，由于多线程是共享内存空间的，因此可以跳过内存空间的切换处理。由于没有发生切换内存空间的动作，因此就不需要改变 CPU 上的内存缓冲（正规叫法是 TLB[12]），这对性能的提升具有很显著的作用。

> [11] 关于多任务切换的详细内容请参考 4.1 节。

> [12] TLB（Translation Lookaside Buffer，转址旁路缓存，也被称为页表缓存、转译后备缓冲器）是为了提高将内存的虚拟地址映射到物理地址这一处理的速度的缓存，是存在于 CPU 内部的结构。一旦发生上下文切换，TLB 就会被刷新（Flash），受此影响若 TLB 缓存失误，则损失的性能成本会比较高。

基于以上两点可以明白如下内容：

- 即使将 **prefork** 变更为 **worker**，对于单位客户端的响应时间也未必会变短

- 即使将 **prefork** 变更为 **worker**，只要拥有足够的内存，则能够同时操作的并发数也并不会增加

- 即使将 **prefork** 变更为 **worker**，只要不存在海量的上下文切换（没有同时并发的大量访问），则改善的效果并不大

因此也请大家认识到将 prefork 切换为 worker 能够改善性能的情况是很有限的。

相反，适合变更为 worker 模式的情况如下：

- 可用内存量不多或仅有少量的内存消耗时。在这种情况下，比进程的内存消耗量少的线程的优点可以得到充分地发挥

- 在上下文的切换次数较多，需要减少该部分的 **CPU** 资源占用时，也就是说大量访问的挤占造成 **CPU** 使用率变高 [13]，需要降低 **CPU** 使用率时。与进程相比，线程间的上下文切换的性能成本较低，因此切换到该模式可以降低 **CPU** 的负载

> [13] 上下文切换次数可以通过 `sar -c` 命令来查询。

一个客户端对应一个进程 / 线程

prefork 和 worker 的共同点是 Apache 对于来自客户端的每个请求，都会分配一个进程或线程来进行处理。也就是说，如果同时有十个客户端发出请求，就会分配十个进程或十个线程来进行响应 [14] 。

> [14] 基于此模式，在 Apache 运行第三方的应用程序时，该应用程序的开发者可以更加容易地进行开发工作。

能够同时生成多少进程 / 线程决定了 Apache 的处理性能，因此需要配置一个最佳的设定项来控制进程 / 线程数，这也可以说是 Apache 优化的关键。让我们继续详细地了解一下吧。

4.2.4　httpd.conf 的配置

httpd.conf 的配置将决定 Apache 的性能，特别会对“能够同时处理的请求数”产生影响，下文将进行详细讲解。

Apache 的安全阀 **MaxClients**

由于 Web 服务器将受理来自数目不详的客户端的请求，因此需要在设计时考量到：无论什么时候遇到什么程度的流量，都应该能够处理。

在此需要根据负载动态控制 Apache 的进程 / 线程数。但是动态控制的结果可能会导致生成大量的进程 / 线程，甚至会占用全部的设备资源。

因此就需要设定 MaxClients 这个安全阀，来设定可以同时连接的请求数的上限值。若没有 MaxClients，当大量请求同时涌来，超过该系统允许的请求数时，将会导致系统内存耗尽，从而致使设备死机，或者 CPU 耗尽变得无法响应等致命故障。

当请求过多时，可以将 MaxClients 设定为：

- 将不能处理完的请求安排在等候队列中等待一段时间
- 若等待队列溢出，对该请求返回错误并将其退回到客户端

以上策略均需要通过 Apache 的安全阀 MaxClients 实现，以避免系统死机等最坏事态的发生。

这个安全阀 MaxClients 的上限值是静态的数值，必须根据设备资源进行手动配置。调整该数值就是 Apache 优化的关键。反过来说，其他项目的调节对性能并没有什么太大的影响。下面就对该安全阀 MaxClients 的数值调整进行详细叙述。

在 **prefork** 模式的情况下

在 prefork 模式的情况下，配置项目相对简单。安全阀就是在 ServerLimit 和 MaxClients 这两个指令中设定的参数。ServerLimit 及 MaxClients 决定了 Apache 能够同时生成的进程数的上限值。这两个参数原本的意义如下。

- **ServerLimit**： 服务器数量，即 **prefork** 中进程数的上限
- **MaxClients**： 能同时连接的客户端数的上限值

但这对使用一个进程处理一个客户端请求的 prefork 来说，两者的意义大致相同 [15] 。要想提高进程数的上限值，可以对 ServerLimit 和 MaxClients 进行配置。一般情况下设定的原则是不能让 MaxClients>

ServerLimit。

> [15] 在 worker 模式等其他 MPM 中，两者的区别具有意义。

```
ServerLimit        50
MaxClients         50
```

因此可以像上面这样先配置 ServerLimit。上面所配置的最大进程数（能够同时连接的客户端的数量）为 50。

除此之外，还有控制进程 / 线程数的 MinSpareServers、MaxSpareServers、StartServers 等参数，由于这些项目对性能的影响并不是很大，因此在这里就不讲解了。

这里还有个问题，即配置该 ServerLimit、MaxClients 时，具体将数值设置为多少才好呢？当然我们不能仅凭感觉来设定这些参数，而要通过

- 服务器所安装的物理内存的容量
- 单位进程的平均内存消耗量

这两点进行估算，以合理配置可以生成多少进程。

第一点通过查看硬件的详细说明即可得知，或者也可以通过 `free` 等命令进行查看。

但第二点，即进程占用的内存大小要如何查看呢？虽然通过 `ps` 和 `top` 也能确认，但是这里还是从 proc 文件系统来查看吧。在 Linux 中，通过`/proc/<进程的PID>/status` 就可以看到进程的内存使用量详情。关于这些项目的意义，请参考内核源代码的附属文件（Documentation/filesystem/proc.txt）。

在图 4.2.3 的摘要中，VmHWM 是该进程实际使用的内存空间的大小。图 4.2.3 的例子是和 mod_perl 模块一起被作为 AP 服务器使用的 Apache 统计，可以了解到目前所使用的物理内存不足 100MB。VmPeak 和 VmSize 是虚拟内存上的空间，在物理内存上所对应的空间由 VmHWM

指示。

```
% cat /proc/23812/status
Name:   httpd
State:  S (sleeping)
<中间省略>
VmPeak:   342544 kB
VmSize:   341036 kB
VmLck:         0 kB
VmHWM:     99016 kB   ←进程实际使用的物理内存区域的大小
VmRSS:     97644 kB
VmData:    94572 kB
VmStk:        84 kB
VmExe:       308 kB
VmLib:     19072 kB
VmPTE:       668 kB
Threads:        1
<以下省略>
```

图 **4.2.3**　进程的内存使用量等摘要

从 VmHWM 的数值可以得知 httpd 各进程的 VmHWM 平均值。例如图 4.2.3 的例子就表示：

- 安装了 **4GB** 的内存容量
- 每个 **httpd** 进程的内存使用量为 **100MB**
- 操作系统的保留内存为 **512MB**
- **4GB－512MB ＝ 3.5GB → 3,500/100＝35**

按照以上逻辑，可以将 MaxClients 设置为 35。

但是，仅这些判断材料还不够。Linux 为了节约物理内存，在父进程和子进程之间还共享了一部分内存。考虑这个共享部分的内存，还可以配置更大的数值。

父子进程共享内存的写时复制技术

所有用户进程都是从其他某个进程 fork 生成的，即所有进程都存在有父进程。在 prefork 模式的 Apache 下，首先启动某个 httpd 父进程，然后该进程再生成多个 httpd 子进程。

在使用 fork 生成进程的情况下，父子进程在不同的内存空间运作，彼此互不干扰。为了实现独立的内存空间，可以使用 fork 从父进程复制整个内存的内容到子进程，但是这个复制处理的成本非常高。

Linux 操作系统中，在进行 fork 操作后，被映射到虚拟内存空间的物理内存空间在不被复制的情况下由父子进程共享。该共享空间是通过分别准备父子进程所需的虚拟内存空间，并从各自的虚拟内存空间映射同样的物理内存空间来实现的（图 4.2.4）。若父进程或子进程对虚拟内存进行写入，那么进行该写入的空间就不能再继续被共享，这时父子进程才分别拥有了映射到该区域的实际的物理空间。

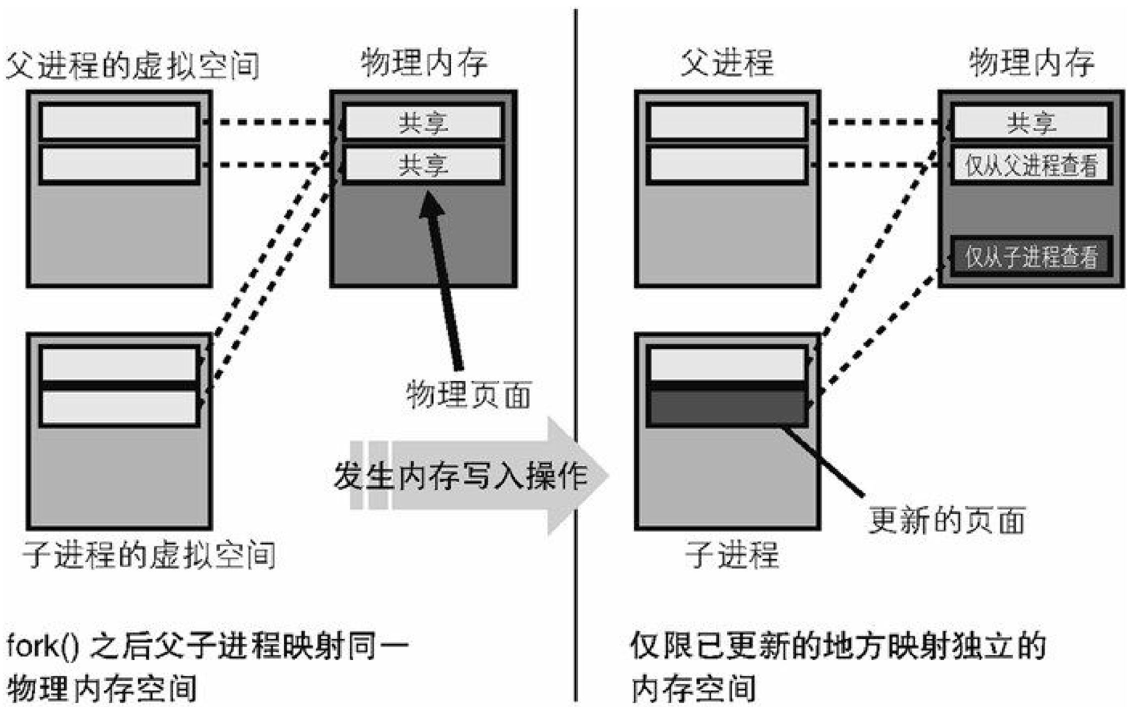


图 **4.2.4**　虚拟内存的写时复制

反过来说，没有进行写入的内存空间能永远被共享，因此就可以避开内存上的页面重复以更加有效地利用内存。

该结构被称为“写时复制技术”（Copy on Write），即“在写入时才进行复制”的意思。也可以理解为 fork 时内存复制的延迟处理。

查看写时复制时共享的内存大小

要配置 MaxClients，还需要考虑在实际使用的内存区域中，父子进程共享的物理内存空间的大小。共享内存空间可以通过 /proc/<进程的 PID>/smaps 的数据查看，但在数据量很多时，调查将变得比较困难。在此制作了调查共享内存大小的 Perl 脚本（代码清单 4.2.1）[16] 。提交动态参数——进程的 ID，就可以查看该进程的共享内存的大小。图 4.2.5 是与 pgrep 配合使用所输出的数据。

> [16] 在 Linux 中执行该 Perl 脚本需要另外安装 Smaps。

代码清单 **4.2.1　shared_memory_size.pl**

```
#!/usr/bin/env perl
use strict;
use warnings;
use Linux::Smaps;

@ARGV or die "usage: $0 [pid ...]";

print "PID\tRSS\tSHARED\n";

for my $pid (@ARGV) {
    my $map = Linux::Smaps->new($pid);
    unless ($map) {
        warn $!;
        next;
    }

    printf
        "%d\t%d\t%d (%d%%)\n",
        $pid,
        $map->rss,
        $map->shared_dirty + $map->shared_clean,
        int((($map->shared_dirty + $map->shared_clean) / $map->rss) * 100)
}
```

```
% shared_memory_size.pl `pgrep httpd`
```

```
PID     RSS

     SHARED


24807   69452   66032 (95%)
24809   76996   55216 (71%)
24810   80812   54292 (67%)
24811   77188   54236 (70%)
24812   79208   54340 (68%)
24813   76608   55492 (72%)
<以下省略>
```

图 **4.2.5**　运行实例

显示的内存大小的单位是 KB（Kilobyte）。RSS 是整个进程分配的内存大小，SHARED 是其中父子进程共享的空间的大小。这里的 70% 左右的数字是父子进程所共享的内存空间的比例。

另外在写时复制技术机制中，随着时间的流逝，该共享的比例会不断下降。启动 httpd 之后，所输出的共享比率的数字自然会升高，因此该数值并没有很大的参考价值。MaxClients 的参数的设定原则是能让系统稳定运行，不发生大的波动。因此可以试着发出一定数量的请求，采用系统相对稳定时的数值即可。然后考虑之前估算过程中父子进程所共享的内存大小，就会得出以下结论：

- 安装了 **4GB** 内存的内存容量
- 每个 **http** 进程的内存使用量为 **100MB**
- 其中 **70%** 被父子进程共享，每个子进程的内存使用量为 **30MB**
- 操作系统的保留内存为 **512MB**
- **4GB－512MB ＝ 3.5GB → 3,500/30＝116.66**

因为平均内存使用量和共享率只是大致计算出来的，所以实际上配置为 100 左右就行了。

MaxRequestsPerChild

在这里做一些补充。基于写时复制的内存共享，共享率会随着时间的流逝而降低。这样的话，在 Web 服务器这样持续运作的软件中，最终大部分的空间都将被持续挤占而不能被共享。

Apache 中会定期结束子进程并建立新的子进程，基于该方法可以避免上述事态的发生。鉴于新建立的子进程是通过父进程的 fork 建立的，因此在建立子进程时，可以将内存中的内容完整地返回到父进程所关联的子进程上。

在此可如下配置 **MaxRequestsPerChild** 指令的参数：

```
MaxRequestsPerChild 1024
```

这里设定了每个进程将会处理 1,024 个请求，该进程处理完第 1,024 次的请求之后就会自动结束，父进程会在此建立新的子进程。

通过合理配置 MaxReqeustsPerChild 的参数，还可以避免使用 mod_perl 及 mod_php 模块运行的应用程序引起内存泄漏，这是发生内存黑洞持续消耗内存时的有效应急措施。

在会接收到大量请求的大型服务器中，如果 MaxRequestsPerChild 的数值太小，就会频繁重复进行进程的建立和结束，因此可能需要适当增加该数值的大小。相反，在请求不多的服务器上，即使将 MaxRequestsPerChild 的数值设置得比较小，此负担也几乎不存在。可以在综合考虑 CPU 负载状况及进程所占用的内存空间大小等基础上，设置出恰当的数值。

在 **worker** 模式的情况下

worker 是多进程和多线程的混合型模式。

- 在一个进程中生成多个线程，一个客户端交由一个线程进行处理
- 生成多个进程

具体行为如上。因此，进程数 × 每个进程的线程数 这一数量的线程将

并发运作。进程的部分使用与 prefork 类似的方法进行优化。而在对线程部分进行优化时，则需要坚持以下原则：

- 线程与进程不同，线程间共享全部的内存空间。不需要考虑像写时复制那样的情况
- 每一个线程需要最大不超过 **8,192KB** 的内存作为栈空间 [17]

> [17] 这是 Apache 的规格。在 Linux 环境中，线程的栈大小由系统指定。此处的 8,192KB 取决于系统，具体可以通过`ulimit -s` 命令进行确认。

在 Worker 模式的情况下，除 ServerLimit、MaxClients 的参数之外，还需要调整 ThreadLimit 和 ThreadsPerChild。worker 模式下的配置需要了解：

- **MaxClients**： 可以在同一时间连接的客户端数，也就是进程数 **x** 线程数
- **ServerLimit**： 最大进程数
- **ThreadLimit**： 每个进程的最大线程数
- **ThreadsPerChild**： 每个进程的最大线程数（与 **ThreadLimit** 大致相同）

MaxClients 是系统能够容许的客户端数量，通过与其他参数配合设定，可以把控进程和线程同时处理的客户端数量。确定 MaxClients 后再确定 ThreadsPerChild 后，就能进一步确定所需的进程数了。例如在 MaxClients 为 4096，ThreadsPerChild 为 128 时，

- **MaxClients 4096/ThreadsPerChild 128 ＝ 32** 进程

因此，需要经常调整以满足 `ServerLimit ≥ MaxClients/ThreadsPerChild` 这个关系。当关系不满足时，这一情况将会被记录在错误日志中。把以上各项加以配置，即：

```
ServerLimit            32
ThreadLimit            64
MaxClients           4096
```

```
ThreadsPerChild       64
```

至于将各参数设定为多少，基本上和 prefork 的情况一样，都要在综合考虑系统所安装的内存容量及每个线程的内存消耗量的基础上进行计算。

若要查看实际在系统上运行了多少条线程，可以在 `ps` 命令上使 用`-L` 选项。正如在 4.1 节中讲解的那样，添加`-L` 参数可以输出 NPTL 的线程，只要数数具体有多少条就 OK 了。

在系统超载的情况下，改变 **MaxClients** 前需要了解**……**

之前提到了“问题只是在 Web 服务器上无法响应的问题表现得很明显，但根本原因未必是 Web 服务器”。其实表面上的问题就是 MaxClients 到达了上限。错误日志中会有如下记录：

```
[Wed Sep 05 17:30:43 2007] [error] server reached MaxClients setting, consi
```

在达到 MaxClients 而不能再生成进程、线程这样的状态下，只不过会警告“发生了某些问题”。虽然也有可能是连接数过多造成 MaxClients 达到上限，但也有可能存在其他的原因。

例如，在将 Apache 作为 AP 服务器使用时，假设其上运行的应用程序要连接到数据库服务器（图 4.2.6）。

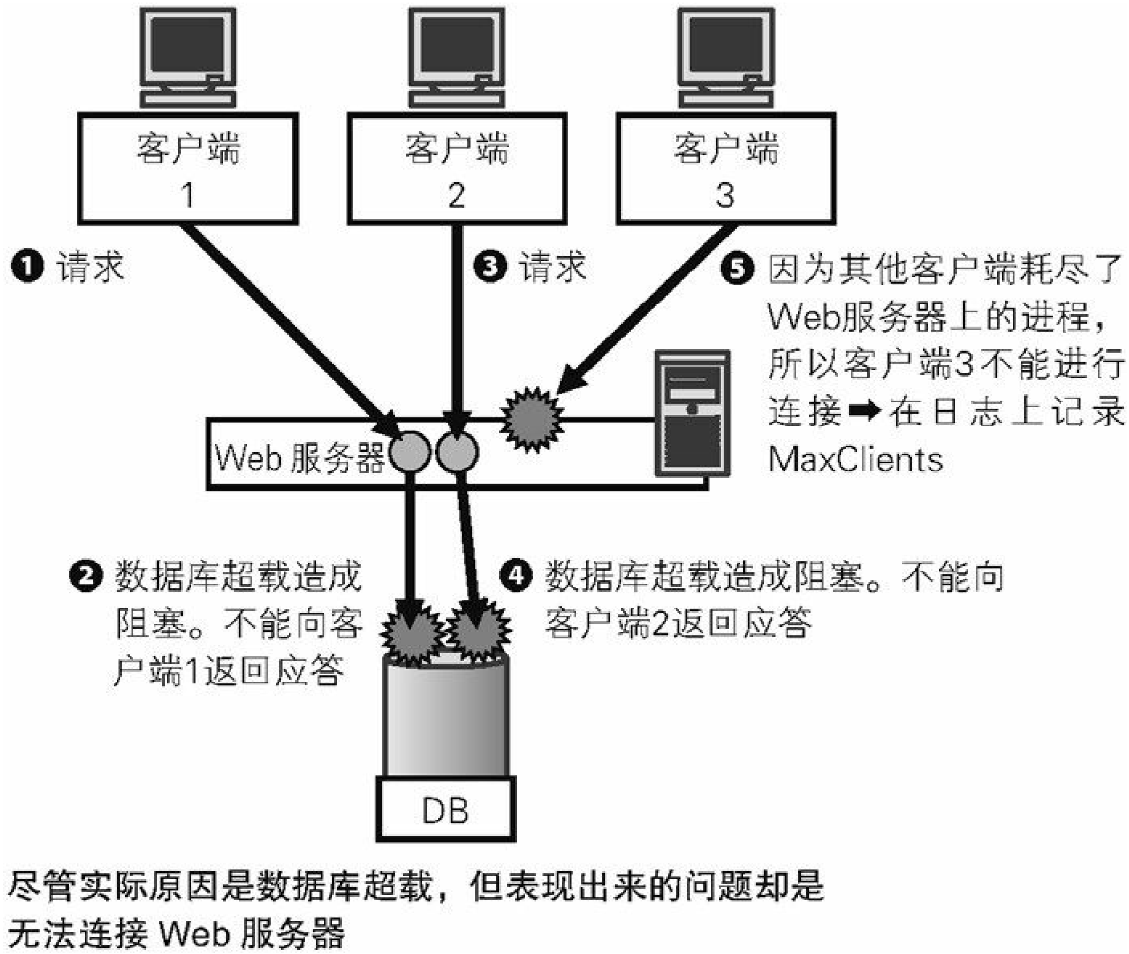


图 **4.2.6**　原因为数据库超载的情况示例

- 如果数据库超载，应用程序将阻塞等待来自数据库的应答
- 结果 **httpd** 进程 / 线程变为被阻塞的状态
- 因为被阻塞的进程 / 线程不能处理来自其他客户端的请求，于是 **Apache** 寻找空闲进程 / 线程
- 如果没有空闲进程 / 线程，则生成新的进程 / 线程
- 如果数据库仍旧超载，新生成的进程 / 线程也会在处理随后的客户端的请求的过程中被阻塞
- 最终 **MaxClients** 到达上限，无法生成新的进程 / 线程

- 将该事件记载到错误日志中

在这种情况下，如何调整 Apache 配置都没有意义。即使增加 MaxClients，未来在客户端发出请求时也会被阻塞，状况依然得不到改善。所以还需要回到根本原因，解决数据库超载的问题。

4.2.5 Keep-Alive

除 MPM 模块的参数以外还有影响性能的参数，比如“Keep-Alive”的配置。Keep-Alive 是在处理完来自特定客户端的请求之后暂时保持连接，以备处理来自相同客户端的其他文件的请求的功能。合理设置后，客户端无需重复连接 / 断开，仅需一次连接就能下载多个文件，客户端 / 服务器并发处理的效率将会显著提高。

另一方面，根据情况的不同，Keep-Alive 也会导致系统发生瓶颈。具体请参考 2.1 节中有关反向代理的讲解。

4.2.6 Apache 以外的选择

虽然本节的中心是讲解 Apache，但是市面上有很多开源且自由的 Web 服务器。虽然 Apache 也是广泛使用的 Web 服务器，但是并不意味着必须使用 Apache。

Apache 的优点之一就是内部完全模块化的通用构造，具备高扩展性。因此，包含第三方模块在内的扩展模块的开发非常繁盛。另外还能自己创建新的模块，自定义 Apache 的运行。将 Apache 作为超越 Web 服务器的网络服务器使用时，比 Apache 更能适用于多种场景的服务器并不多见。

另一方面，Apache 的性能怎么样呢？ Apache 目前采用多进程 / 多线程模式。除此之外的网络服务器的代表模式还有单进程·事件驱动（Single Process Event Driven，**SPED** ）模式。在 SPED 模式的服务器上，不是通过多个运行单位来处理多个连接，而是利用系统的功能，让单一进程监控多个网络连接的输入输出事件，并结合输入输出事件将处理快速切换，以实现高效的并发处理。

单纯从多线程和 SPED 的结构来看，并没有好坏之分。另一方面，以安装的广泛度来讲，Apache 的确是最普及的 Web 服务器。但在一个请求

周期内，处理请求所需的 CPU 计算量、内存消耗量较大，而有多少个进程 / 线程，就会相应地消耗相当大程度的资源，这是它的缺点。

lighttpd

最近开源 Web 服务器中比较受欢迎的是 **lighttpd** [18] 。lighttpd 具备以下特点：

> [18] ***URL*** http://www.lighttpd.net/

- 采用 **SPED**，通过少量内存并发处理大量访问，处理速度让人满意
- 虽然相较于 **Apache**，**lighttpd** 的通用性较差，但是因为其单位请求所需的计算量少，可以降低 **CPU** 负载
- 单位进程的内存消耗量远远小于 **Apache**
- 涵盖了相当于 **Apache** 的核心模块、**mod_rewrite** 和 **mod_proxy** 模块的基本功能
- **支持 FastCGI，可以优化用 Perl 及 PHP、Ruby 所编写的 Web 应用程序，可以被作为 AP 服务器使用

因此 lighttpd 在大规模并发环境中的运行效果也很好。Hatena 网站现在已经将一部分的 Apache worker 更换为了 lighttpd。

比较 lighttpd 和 Apache，最显著的差异是内存的消耗量。lighttpd 无论有多少连接，都能用一到几个进程完成所有处理 [19] 。因为 lighttpd 是根据客户端数来增减进程 / 线程数量的，这是 lighttpd 与 Apache 决定性的差异。

> [19] 使用 select(2)/poll(2) 和 epoll 等的文件描述符监控系统调用，实现多网络 I/O 的并发处理。

lighttpd 适用于传送大量静态文件。在需要将大量文件返回到大量客户端时，也可以以最小限度的资源消耗去完成。

当然 lighttpd 也可以传送动态的网页内容。由于 lighttpd 的使用相当简

单，因此有很多基于 lighttpd+FastCGI 环境，来高速运行用脚本语言开发的 Web 应用程序的事例。但是，在用 lighttpd 传送动态内容时，Apache+mod_perl（mod_php 等）与 lighttpd+FastCGI 的组合并没有显著的性能差异 [20] 。

> [20] 由于 Apache 具备丰富的 API，可以用来自定义应用程序，因此笔者经常使用 Apache。

虽然这里没有对 lighttpd 进行详细讲解，但想以更少的资源处理大量客户端的连接时，可以考虑使用 lighttpd。

4.3　MySQL 的调优诀?

4.3.1　MySQL 的调优诀窍

当数据库服务器要求更高的性能该怎么办呢？单刀直入，用一句话说就是“如何快速存取数据”。

那么数据库服务器的性能调优，也就是“想要以更短的时间存取数据”，需要具体考虑什么策略呢？这根据调优的视角可以分为几类，以下首先基于调优的视角进行简单的整理。

基于调优视角的分类

首先，可以考虑从以下几个视角进行分类：

1　服务器方面

2　服务器之外的其他方面

3　周边系统方面

1 服务器方面

第一个是“服务器方面的调优”。说到服务器方面的调优，首先需要了解“mysqld 的参数优化”。特别是内存关联的参数以及磁盘 I/O 关联的参数是调优的关键。

除 mysqld 的参数以外，还包括系统方面的优化，例如：

- 对磁盘 **I/O** 关联的内核参数进行调整
- 合理利用文件系统及 **mount**（挂载）命令的选项等

本节中也将其归类为服务器方面的优化。

参数以外的其他方面的优化，还有“分区技术”（Partitioning）。如果数据规模变大，数据大小和访问量增加，仅使用一台数据库服务器通常会无法进行处理。

于是，在此以表为单位分割数据库服务器，或者将表格数据以主键（Primary Key）等为单位切分到数据库服务器上。据此就可以将数据切割为较小的单位以便轻松缓存，通过分流访问还可以降低单台服务器的负载。另一方面，由于需要在被分割的数据库服务器集群里查找出相应的数据以供处理，以及在 SQL 语句层面关系表将不能被关联，因此在应用程序方面的负载会有所增加。

2 服务器之外的其他方面

第二点是除服务器之外的其他方面的优化。具体指如下事项：

- 表的设计

  → 创建合适的索引

  → 根据需要可能会进行非标准化处理

- **SQL** 的优化

  → 很好地使用索引

  → 调整表关联的顺序、方法

特别是 SQL 的优化，由于部分执行时间较长的语句可以通过慢查询记录（log-slow-queries）找出，再通过 MySQL 的 EXPLAIN 语句及相关工具弄清其原因，所以优化过程还是比较容易的。

3 周边系统

最后是“周边系统的调优”。首先什么是周边系统的调优呢？在文章的开头提到了调优的目标是“以更短的时间存取数据”。需要注意的是在此是以访问数据库服务器为视点的，若访问周边系统存取数据的速度比访问数据库服务器更快，则不需要特意去访问数据库服务器。

举个具体的例子，在访问数据的客户端和数据库服务器之间增加 memcached 等缓存服务器，这样就可以通过访问缓存服务器来调用数据，而不用去访问数据库服务器了。

说到 RDBMS 的优化，许多情况下往往只会想到 SQL 语句和服务器参数的优化。但如果将这些作为“输入输出数据的一个系统”，将客户端和数据库服务器看作这个系统的构成要素，那么就可以考虑通过增设缓存服务器来优化该系统的性能，而并非改动具体的参数。大家一定要具备这种宏观的视角。

本章接下来要处理的内容

截至目前，我们将优化的视角分为了三类并分别进行了讲解，下一阶段将简述实际的调优工作，即发现瓶颈 → 采取各种策略消除瓶颈。

瓶颈的原因潜藏在各个地方。因此瓶颈的发现，并不是单纯的“发现慢的 SQL 语句就行了”，最好先以上述三个视角进行宏观的监控。

虽然这么说，但由于瓶颈的产生多起因于条件及 RDBMS 的用法问题，存在各种不同的情况，不能具体断定“就是某处存在问题”。

另外，在对数据库和表进行分区以及引入缓存服务器之前，需要优先研究一下 SQL 语句的结构以及进行参数的优化，以最大可能地发挥数据库服务器的性能，如果这样还不能解决问题，其后才应该考虑进行分区或引入缓存服务器。

在本节之后，我们将继续把焦点放在服务器方面的优化上，对 MySQL 服务器（mysqld）的参数调节进行深入讲解。

本章中所讲述的 MySQL 的版本是 5.0.45。

4.3.2　内存相关的参数优化

在有关 MySQL 服务器的优化中，十分重要的是内存（缓冲）相关的参数，下面介绍以下两点：

- 优化的要点
- 由于这里的数据库服务器是在内存为 **4GB** 的情况下进行相应的配置设定的，因此具体数据仅供参考

缓冲的种类 **……**优化时的注意事项 ❶

首先需要注意的地方是，MySQL 中为了提高性能开辟了暂时储存数据的内存空间，通常被称为缓冲，缓冲有以下两个类型：

- 全局缓冲（**Global Buffer**）
- 线程缓冲（**ThreadBuffer**）

所谓全局缓冲，是 mysqld 内部的唯一一个缓冲。相应的，线程缓冲则是针对每个线程（连接），都确保一个缓冲。

优化参数时，需要认识到全局缓冲和线程缓冲的差异。这是因为，如果给线程缓冲分配过多的内存，那么一旦连接增加，内存很快就会不足。

不能分配太多 **……**优化时的注意事项 ❷

给缓冲分配的内存越大，性能就越高。话虽如此，如果分配的数值超过了服务器的物理内存容量，造成频繁访问交换区，实际性能反而会降低。

比起在 MySQL 层面进行参数优化，在某些情况下，通过使用 MyISAM 的数据引擎，将数据库文件放到系统的磁盘缓冲上，这样更能提高 MySQL 表的性能。

内存的相关参数

内存的相关参数如表 4.3.1 所示。

表 **4.3.1**　内存相关的参数

参数
说明
innodb_buffer_pool_size
用途 innoDB缓冲区，用来存放数据和索引 缓冲类型 全局缓冲　参考值 512MB（一般设置为机器内存的50%-80%） 因为是全局缓冲，所以推荐分配得宽裕一些
innodb_additional_mem_pool_size
用途 innoDB内存池，存放着各种内部使用的数据 缓冲类型 全局缓冲　参考值 20MB 没有大量分配的必要。不足时会通过错误日志发出警告，届时再增加也没有问题
innodb_log_buffer_size
用途 InnoDB日志缓冲区 缓冲类型 全局缓冲　参考值 16MB 基本上是8MB，最多为64MB，不需要太大。缓冲Buff是每当数据库事务被提交（Commit）时进行的，几乎每秒都会发生缓存到磁盘的动作，因此建议将所需的其他参数分配得大一些
innodb_log_file_size
用途 InnoDB日志文件，存储在磁盘上。虽然它不是内存，但却是重要的优化参数 缓冲类型 ---　参考值 128MB 数值越大性能越高。详情请参照正文
sort_buffer_size
用途 被用于ORDER BY和GROUP BY的内存空间 缓冲类型 线程缓冲　参考值 2MB 由于是线程缓冲，因此要注意过度增加会导致内存不足。笔者建议设为2MB或4MB
read_rnd_buffer_size
用途 在排序后读取结果数据时使用的缓冲区。因为磁盘I/O会有所减少，所以可以提高ORDER BY的性能 缓冲类型 线程缓冲　参考值 1MB 因为这个也是线程缓冲，所以也需要注意不要分配过多。笔者建议设为512KB～2MB
join_buffer_size
用途 不使用索引的表关联时使用的内存空间 缓冲类型 线程缓冲　参考值 56KB 线程缓冲。由于仅在没有索引时才进行关联操作，因此从性能提高的观点上对该关联表的期望不大，所以该参数不需要很大
read_buffer_size
用途 对数据表顺序扫描的缓冲大小 缓冲类型 线程缓冲　参考值 1MB 如果这里也考虑性能，就应该使用类似索引的查询，因此不需要那么大的空间
key_buffer_size

用途 缓存MyISAM的键（索引）的内存空间 缓冲类型 全局缓冲　参考值 256MB 因为是全局缓冲，所以为了提高性能可以分配多一些。如果不（太）使用MyISAM，可以将该值设置得小一些，以将内存分配给其他参数
myisam_sort_buffer_size
用途 MyISAM中以下情况的索引排序时所使用的缓冲空间 • REPAIR TABLE　• CREATE INDEX　• ALTER INDEX 缓冲类型 线程缓冲　参考值 1MB 因为通常在语句（DML）中不大使用，所以不需要那么多

下面对表 4.3.1 进行补充说明。首先，关于 innodb_log_file_size，mysqld 在 innodb_log_file 存满时，仅将内存上更新的 innodb_buffer_pool 部分写入磁盘上的 InnoDB 数据文件。因此，如果只增加 innodb_buffer_pool_size，而不同时调整这个 innodb_log_file_size 的话，innodb_log_file_size 马上就会溢出，进而就不得不频繁地进行 InnoDB 数据文件的写入处理，最终就会导致性能下降。

innodb_log_file_size 的数值要设置为 1MB 以上，32bit 的设备的情况下必须设置为 4GB 以下，具体在 MySQL AB 文档中有详细描述。

这里还有一个上限。虽然 innodb_log_file 只设置 innodb_log_files_in_group 的数量（默认 2），但要确保 innodb_log_file_size×innodb_log_files_in_group 不超过 innodb_buffer_pool_size。

综上，可以得出如下结论：

$$1\text{MB} < \text{innodb_log_file_size} < \text{MAX_innodb_log_file_size} < 4\text{GB}$$

$$\text{MAX_innodb_log_file_size} = \frac{\text{innodb_buffer_pool_size}}{\text{innodb_log_files_in_group}}$$

其他必须注意的是，innodb_log_file_size 的值越大，InnoDB 的崩溃恢复所需的时间就越长。

其次，这里也对表 4.3.1 中的 key_buffer_size 稍作补充说明。关于 Key Buffer 的命中率，可以使用`SHOW STATUS`的数值，用下面的公式算出：

Key Buffer 的命中率 = `100-(key_reads/key_read_requests×100)`

4.3.3 内存相关的检查工具 ……mymemcheck

最后介绍一下笔者正在使用的自制工具 mymemcheck。mymemcheck 可以基于 my.cnf 或者`SHOW VARIABLES` 的结果，进行以下三个方面的检查：

- 至少所需要的物理内存大小
- **IA-32 Linux** 中堆的大小范围
- **innodb_log_file_size** 的最大值

以上都是 MySQL AB 的文档中所罗列的事项，由于内存相关的参数也有彼此相关的，不注意的话配置的数值会相互矛盾。因此，在变更参数时，可以使用该 mymemcheck 工具确认所设置的数值是否合理。

运行结果如图 4.3.1 所示。

```
$ ./mymemcheck my.cnf

[ minimal memory ]
ref
  * 『High Performance MySQL』, Solving Memory Bottlenecks, p125

global_buffers
  key_buffer_size                       268435456   256.000 [M]
  innodb_buffer_pool_size               536870912   512.000 [M]
  innodb_log_buffer_size                 16777216    16.000 [M]
  innodb_additional_mem_pool_size        20971520    20.000 [M]
  net_buffer_length                         16384    16.000 [K]

thread_buffers
  sort_buffer_size                        2097152     2.000 [M]
  myisam_sort_buffer_size                 1048576  1024.000 [K]
  read_buffer_size                        1048576  1024.000 [K]
  join_buffer_size                         262144   256.000 [K]
  read_rnd_buffer_size                    1048576  1024.000 [K]

max_connections                               250

min_memory_needed = global_buffers + (thread_buffers* max_connections)
```

```
                  = 843071488 + 5505024* 250
                  = 2219327488 (2.067 [G])

[ 32bit Linux x86 limitation ]
ref
  * http://dev.mysql.com/doc/mysql/en/innodb-configuration.html

  * need to include read_rnd_buffer.
  * no need myisam_sort_buffer because allocate when repair, check alter.

          2G > process heap
process heap = innodb_buffer_pool + key_buffer
               + max_connections* (sort_buffer + read_buffer + read_rnd_buf
               + max_connections* stack_size
             = 536870912 + 268435456
               + 250* (2097152 + 1048576 + 1048576)
               + 250* 262144
             = 1919418368 (1.788 [G])

          2G > 1.788 [G] ... safe

[ maximum size of innodb_log_file_size ]
ref
  * http://dev.mysql.com/doc/mysql/en/innodb-start.html

  1MB < innodb_log_file_size < MAX_innodb_log_file_size < 4GB

MAX_innodb_log_file_size = innodb_buffer_pool_size* 1/innodb_log_files_in_g
                         = 536870912* 1/2
                         = 268435456 (256.000 [M])

    innodb_log_file_size < MAX_innodb_log_file_size
               134217728 < 268435456
             128.000 [M] < 256.000 [M] ... safe
```

图 **4.3.1　mymemcheck** 的运行实例

第 5 章　高效运行 ——确保服务的稳定提供

5.1　服务状态监控 Nagios

5.1.1　稳定的服务运营与服务状态监控

服务状态监控是稳定的服务运营所不可或缺的。即便服务器已经做了冗余的处理，由于一时疏忽，或者冗余的设备出现故障，都会造成很严重的情况。若故障再度出现，服务就很有可能停止运作。当系统的某个部分出现异常时，通过服务状态监控及时知晓，对维持服务的稳定运行是很重要的。

Nagios [1] 是著名的开源服务状态监控工具。由于 Nagios 配置灵活，因此在世界范围内被广泛使用。

> [1] ***URL*** http://www.nagios.org/

5.1.2　状态监控的种类

通常情况下，服务状态监控并不仅仅是监控服务的某个功能是否正常运行，还包含确认负载状态等方面。服务的状态监控可分为以下三类：

1 主机或服务的运行状态等网络服务存活状态的监控

2 主机的 **CPU** 占用率或服务的吞吐量等负载状态的监控

3 在一段时间内（一个月或是一年）服务正常运行的时间所占的比例，即可用率的统计

1 存活状态的监控

存活状态的监控是指确认某个功能是否可用，是基本的状态监控操作。

比如，通过使用 ping 指令确认主机是否在正常运作、服务的 TCP 连接是否可用、目标服务器是否能够进行基本的协议处理，从而检查服务状态是否正常。

如果通过 ping 指令没有接到目标服务器的应答，或者 TCP 连接不可用、无法进行基本的协议处理，就能断定目标主机或服务已经处于不可用的状态，这时监控系统会及时通知运维人员。运维人员接到通知后，会重启主机或服务，或启用事先准备好的备用服务器等，以及时解决故障。

在被监控的服务已经进行了冗余处理的情况下，状态监控就并不仅针对提供服务的主机，通过同时监控经过冗余处理的 VIP（虚拟 IP 地址），就可以从最终用户的视角来判断服务是否还在正常运作。这样当故障发生时，就能轻易判断出经过冗余处理的主机的某处故障会不会影响到服务，是否会对服务的正常提供造成影响。

在图 5.1.1 的例子中，即便服务器 B 出现了故障，负载均衡器也会自动将请求切换到服务器 A，服务丝毫不受影响。在这样的架构下，不仅要监控服务器 A 与服务器 B，还需要将负载均衡器的 VIP 也列为监控对象，据此来确认是否会对服务造成影响，并判断故障的紧急程度。

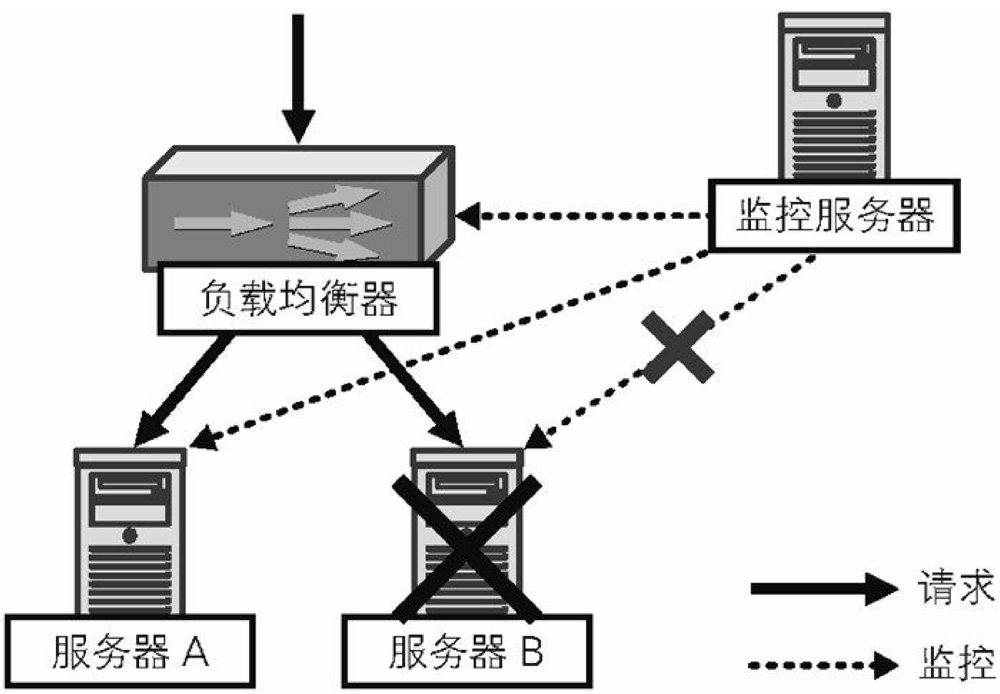


图 **5.1.1**　经过冗余处理的情况下的监控

2 负载状态的监控

负载状态的监控是指检查是否存在一些情况“虽不会令服务停止运作，但会对服务的运行带来过大压力”，是一种非常实用的监控。

为了监控负载状态，可以从统计目标主机的 CPU 负载、当前系统中等待进程的数量等信息着手，判断目标主机的负载是否已经达到了异常的程度[2]。另外，还可以统计在提供服务的进程的请求队列中等待的请求数，以及请求的响应时间，来监控在所允许的服务等级的情况下，向该服务的进程发出的请求是否能够得到处理。若是出现了等待进程数异常多，或响应时间过长的情况，那就能断定主机或服务已经出现了负载过高的问题。

> [2] 关于统计负载，在 4.1 节有详细说明。

负载过高的原因可分为下列三种情况：

❶ **DoS** 攻击等造成的异常请求导致负载过高

❷ **Slashdot** 效应[3]等造成的突发请求导致负载过高

> [3] Slashdot 效应指的是当一个受众广泛的站点引荐了另一个小众的网站后，该小众网站访问人数激增的情况。

❸ 纯粹因为服务太受欢迎，请求数平稳提升而导致负载过高

因此，负载过高的应付办法也比较复杂，需要针对各种不同的负载问题，找到具体的解决方法。例如，在 ❶ 的情况下，要提前预知流量异常的情况，并及时截拦请求；在 ❷ 的情况下，需要设置高速缓存；而在像 ❸ 那样因为服务自身太受欢迎而请求数增加的情况下，就需要考虑扩充主机等了。

通过监控负载状态，能够检测出“服务能够使用，但是运行较慢”的情况，但这种情况单纯通过存活状态的监控是检测不出来的。因此需要对这种情况加以应对，以维持良好的系统响应。

3 可用率的统计

可用率的统计与上述两种监控不同，它是通过分析数周甚至数月间某方面的监控结果，发现并改善系统在长期运行中存在的问题。

通过存活状态的监控和负载状态的监控，可以得知服务的具体运作情况以及负载情况，从而客观地分析系统的哪个部分容易出现问题，系统整体的可用率是怎样的等。

基于这一分析，我们就能够掌握特定主机的不稳定性，或者了解到系统架构原本就是不稳定的，并据此战略性地考量整个系统的冗余状况、运维人员的维护习惯等。

5.1.3 Nagios 概述

Nagios 是具有以上介绍的监控和统计功能的代表性的监控工具。Nagios 可以通过`ping` 指令测试网络服务的存活状态，通过 TCP 连接进行各种服务的监控，通过 SNMP（Simple Network Management Protocol）进行主机的状态监控，甚至还可以通过插件进行任意方式的监控。另外在监控结果的通知方面，除了基本的邮件通知方式外，还可以任意设定其他方式。而且通过 Web 接口，能方便地参考监控目标的状态，还能控制监控的停止或开始。

我们以本节写作时（2008 年）的最新版本 Nagios 3.0.2 为基础来说明。

安装 **Nagios**

由于 Red Hat Enterprise Linux 5 及 CentOS 5 的标准软件包内没有包含 Nagios，因此请从下面的官方网址下载软件包来安装。

URL http://www.nagios.org/

将 Nagios 安装完成后，还可以安装“Official Nagios Plugins”脚本以监控更多的对象。Official Nagios Plugins 中还有其他的环境依赖包，需要时可根据情况添加安装。本节的说明均以在 CentOS 5.0 系统中进行的操作为准。

5.1.4 Nagios 的配置

由于 Nagios 的配置更具弹性，因此配置起来也稍微有些复杂。可以参

考安装路径内的示例配置文件来进行配置。

首先，配置 Nagios 所必需的基本概念请参考表 5.1.1。接下来，我们对表 5.1.1 中的主要概念进行一些说明。

表 **5.5.1　Nagios** 的基本概念

术语	说明
host（主机）	服务器或路由器等网络上的基本的物理元素，即主机
hostgroup（主机组）	将多个主机进行分组，每组至少拥有一个主机。可以以主机组为单位，指定各主机的事件（主机故障、恢复等）的通知对象
service（服务）	主机上所运行的服务。该服务不仅支持POP及HTTP等功能，还可以返回ping响应及空余的磁盘容量等
servicegroup（服务组）	将一个以上的服务进行分类，以方便显示在Web管理界面上
contact（通知对象）	定义Nagios中的各种事件的通知对象（contact，即通知发往的目的地）
contactgroup（通知对象组）	将多个通知对象进行分组。主机组及服务中指定的通知对象就是通知对象组
template（模板）	在多个主机及服务的设定存在公共部分的情况下，可以通过使用模板，简化公共部分的设定，让设定的参数列表更加简洁

配置文件

在下面的解说中，我们假设在安装 Nagios 3.0.2 的同时，还一并安装了“nagios.cfg”“commands.cfg”“localhost.cfg”等配置文件。

配置文件是在多个扩展名为 cfg 的文件中存储的，这么设计是为了方便在别的文件中嵌套读取这些配置项目。随着主机数的增加，配置文件也会变得越来越庞大，为了查阅方便，可以将配置文件进行分割。

host ……主机的配置

通过 host 对需要监控的主机进行必要的配置。由于通过 host 设定的项目多种多样，并且这些设置项目在多个主机中存在很多共同之处，所以建议使用模版。

代码清单 5.1.1 中的例子是，首先定义了 generic-host 的模版，然后基于该模版定义了主机 localhost。

各配置项目在代码清单 5.1.1 中的注释中有简单的说明，接下来对重要的配置项目进行补充说明。

- **flap_detection_enabled**

  发生故障和修复故障这两种行为频繁地重复出现，这种现象就称之为 Flapping。一旦 Flapping 发生，就会涌来大量的通知信息，此时别的有用的通知信息就可能会被淹没。将该配置参数设置为有效时，如果监控出了 Flapping，就会只发出开始和结束的通知

- **max_check_attempts**

  当检测失败的次数超过这里指定的次数时，就认为主机发生了故障，并发出通知

- **notification_period**

  发送通知的时间段。localhost.cfg 中有“24×7”（24 小时）、“workhours”（工作日上午 9 时到下午 5 时）、“nonworkhours”（工作日上午 9 时到下午 5 时以外的时间）、“none”（没有对应的时间段）这四种定义方式。一般使用“24×7”的配置方法

- **check_command**

  监控时使用的命令。配置例子中指定的 check-host-alive 是基于通过

发出ping 对主机进行监控的check-ping 命令的命令。默认的配置是如果 5000 毫秒以内没有做出反映，就会变成 CRITICAL 状态

service ……服务的定义

service 定义了目前主机上所运行的服务（代码清单 5.1.1）。和 host 一样，也可以使用模版功能使描述更加简洁。

代码清单 **5.1.1　host** 的配置案例

```
define host{
  name                            generic-host ←模版名
  notifications_enabled           1 ←开启主机的通知
  event_handler_enabled           1 ←开启主机的事件处理程序（Event Handler）
  flap_detection_enabled          1 ←监控Flapping
  failure_prediction_enabled      1 ←开启故障预判
  process_perf_data               1 ←处理有关性能的信息
  retain_status_information       1 ←重启后依然保留状态信息
  retain_nonstatus_information    1 ←重启后依然保留状态以外的信息
  notification_period             24x7 ←在所有时间段都会通知
  register                        0 ←这里默认即可
}
define host{
  name                            linux-server ←模板名
  use                             generic-host ←指定所使用的模板
  check_period                    24x7 ←在所有时间段都会监控
  max_check_attempts              10 ←规定最多监控10次
  check_command                   check-host-alive ←监控用的命令
  notification_period             workhours ←只在工作日白天通知
  notification_interval           120 ←通知间隔
  notification_options            d,u,r ←通知的状态
  contact_groups                  admins ←通知对象所在的contactgroup
  register                        0 ←这里默认即可
  }
define host{
  use          linux-server ←指定使用的模板
  host_name    localhost ←主机名
  alias        Localhost Server  ←别名
  address      192.168.0.1   ←IP地址
}
```

command ……命令的定义

可以用 **command** 定义命令。在代码清单 5.1.2（service 的配置案例）中，用`check_ping` 对服务进行了监控。命令名（`check_ping` ）后面的`"!100.0,20%!500.0,60%"` 是该命令的参数，用 ! 隔开。这里`"100.0,20%"` 对应了 $ARG1$ ，`"500.0,60%"` 对应了 $ARG2$。

这个命令的默认设定如代码清单 5.1.3 所示。可以使用 `-w` 参数来定义发生 WARNING 的条件，使用 `-c` 参数来指定 CRITICAL 的条件。在代码清单 5.1.2 的情况下，若出现 100ms 以上的延迟或者 20％以上的丢包的话，就会发生 WARNING ；若出现 500ms 以上的延迟或者 60％以 上的丢包的话，就会发生 CRITICAL。

代码清单 **5.1.2　service** 的配置案例

```
define service{
  name                              generic-service ←模板名
  active_checks_enabled             1 ←启动主动监控
  passive_checks_enabled            1 ←启动被动监控
  parallelize_check                 1 ←在主动监控中启动并行监控
  obsess_over_service               1 ←在分流环境中通知结果
  check_freshness                   0 ←关闭强制刷新（freshness监控）
  notifications_enabled             1 ←开启主机的通知
  event_handler_enabled             1 ←开启事件处理程序
  flap_detection_enabled            1 ←启动抖动监控
  failure_prediction_enabled        1 ←开启故障预判
  process_perf_data                 1 ←处理有关性能的信息
  retain_status_information         1 ←重启后依然保留状态信息
  retain_nonstatus_information      1 ←重启后依然保留状态以外的信息
  is_volatile 0
  register            0 ←这里默认即可
}
define service{
  name                              local-service ←模板名
  use                               generic-service ←指定所使用的模板
  check_period                      24×7 ←在所有时间段都会监控
  max_check_attempts                4 ←最多尝试4次监控
  normal_check_interval 5
  retry_check_interval 1
  contact_groups admin
  notification_options              w,u,c,r ←通知的状态
  notification_interval             60 ←通知间隔
  notification_period               24×7
  register            0
```

```
}
define service{
  use local-service  ←指定所使用的模板
  host_name localhost   ←所使用的主机组
  service_description PING  ←服务名
  check_command check_ping!100.0,20%!500.0,60%    ←监控命令
}
```

代码清单 **5.1.3　command** 的配置案例

```
# 'check_ping' command definition
define command{
        command_name    check_ping
        command_line    $USER1$/check_ping -H $HOSTADDRESS$ -w $ARG1$ -c $A
        }
```

contact 与 **contactgroup ……**通知对象和通知对象组

可以用 **contact** 定义通知对象，用 **contactgroup** 定义通知对象组（代码清单 5.1.4）。通知通常都是通过指定通知对象组进行，因此最少要有一个通知对象组。

可以使用 service_notification_commands 命令来处理服务相关的通知，用 host_notification_commands 命令处理主机相关的通知。代码清单 5.1.4 是配置邮件通知的实例。

代码清单 **5.1.4　contact** 及 **contactgroup** 的配置案例

```
define contact{
  contact_name                  nagios-admin ←通知对象名
  alias Nagios                  Admin ←通知对象的别名
  service_notification_period   24×7 ←处理服务通知的时间段
  host_notification_period      24×7 ←处理主机通知的时间段
  service_notification_options  w,u,c,r ←服务通知的事件类型
  host_notification_options     d,r ←主机通知的事件类型
  service_notification_commands notify-by-email ←服务的通知命令
  host_notification_commands    host-notify-by-email ←主机的通知命令
```

```
  email                            nagios-admin@localhost ←通知对象的邮箱地址
  }

define contactgroup{
  contactgroup_name       admins ←通知对象组的名字
  alias                   Nagios Administrators ←通知对象组的别名
  members                 nagios-admin    ←该通知对象组中所包含的通知对象名
  }
```

配置的测试

更改 Nagios 的配置的时候，可以用以下命令让配置立刻生效：

```
/etc/init.d/nagios reload
```

假设配置中存在语法错误等问题，执行以上命令时将会输出这些错误信息。可以根据情况对配置文件进行恰当的修正。

5.1.5 Web 管理界面

Nagios 拥有强大的 Web 管理界面，能够在此确认各种主机和服务的状态，还可以暂停监控等进行某种程度的控制。图 5.1.2 是 Nagios 的主菜单。关于菜单的内容在表 5.1.2 中进行了说明。

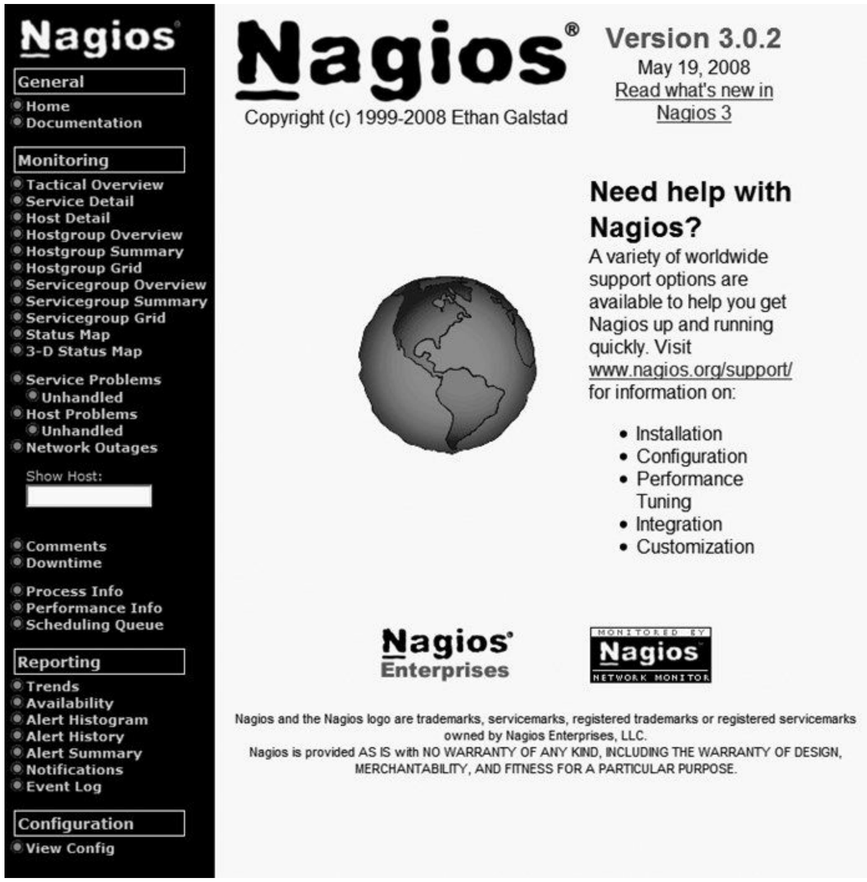


图 **5.1.2　Nagios** 的主菜单

表 **5.1.2　Nagios** 的主菜单

项目	说明
Monitoring	Monitoring项目下从各种角度列出了监控相关的状态
Tactical Overview	列出每个主机及服务的状态信息，还能显示出各个主机及服务的监控情况，以便把握系统整体的健康信息
Service Detail	列出不同主机的服务详情。可以通过对报表进行排序以从整体上把握
Host Detail	通过浏览主机的详情，可以了解该主机是否已经遭遇问题

Hostgroup Overview	确认各个主机组的状态。通过组的划分可以从不同的目的进行把握。另外通过主机组名的链接，可以只查看特定组的信息
Hostgroup Summary	列出各主机组的状态情况及数目
Hostgroup Grid	类似Hostgroup Overview，以表格的形式列出，方便在故障时特定到具体的服务（图5.1.3）
Servicegroup Overview Servicegroup Summay Servicegroup Grid	将Hostgroup Overview、Hostgroup Summay、Hostgroup Grid的主机组划归到相应的服务组时的界面
Status Map 3-D Status Map	可以查看主机的网络拓扑关系图[4] 。Status Map是2D图，3-D Status Map是根据VRML创建的3D图表
Service Problems	只列出发生故障的服务
Host Problems	只列出发生故障的主机
Network Outages	列出在拓扑中发生故障时网络中断的具体信息
Comments	显示主机及服务的相关备注信息
Downtime	列出计划中的停机时间
Process Info	Nagios的进程信息（只有管理员才能看到）。除了显示进程的启动时间等信息外，还能暂时停止Nagios的监控和通知

Performance Information	根据Nagios所监控的性能信息，显示执行监控命令的时间的统计等信息
Scheduling Queue	显示高度队列信息（只有管理员才能看到），列出相关进程下次监控的时间序列
Reporting	在此菜单可以查看之前的监控结果
Trends	生成关于主机或服务的状态变化的趋势
Availability	生成主机或服务在一定时期内的正常运行率及各状态的频率，以及列出监控日志
Alert Histogram	生成关于主机或服务的警告数的变化趋势
Alert History	显示所有主机及服务的警告的历史记录（图5.1.4）
Alert Summary	从指定的主机和服务的警告记录中，显示最新的25条，或最多显示25条
Notifications	查看过去发出的通知，还能通过特定的种类分类浏览这些通知
Event Log	显示Nagios的启动、警告、通知等各种事件日志
Configuration	配置菜单
View Config	列出主机及服务的配置情况

[4] 该图像仅限在 Windows 操作系统中的 IE 浏览器下，安装 cortbeta 才会显示。——译者注

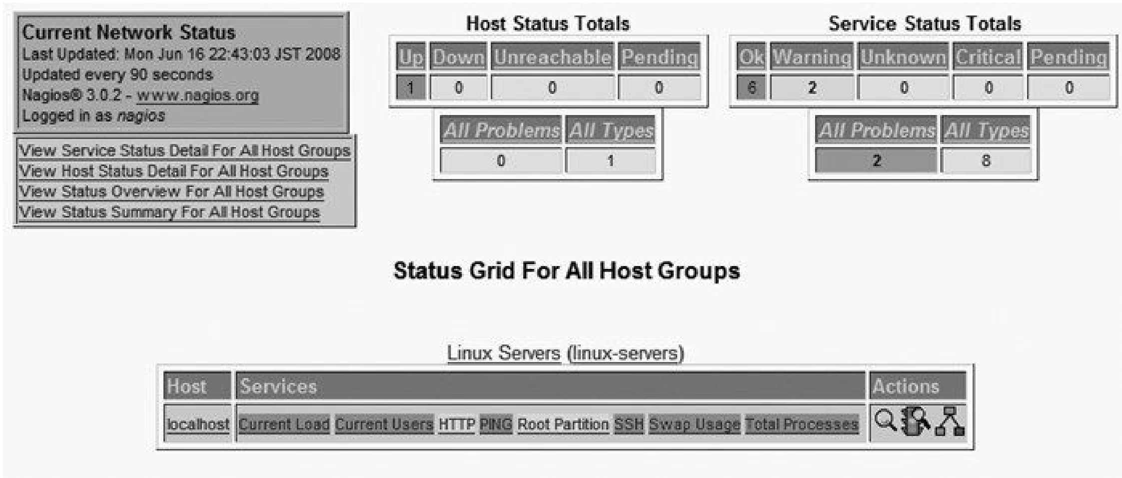


图 **5.1.3　Hostgroup Grid** 界面

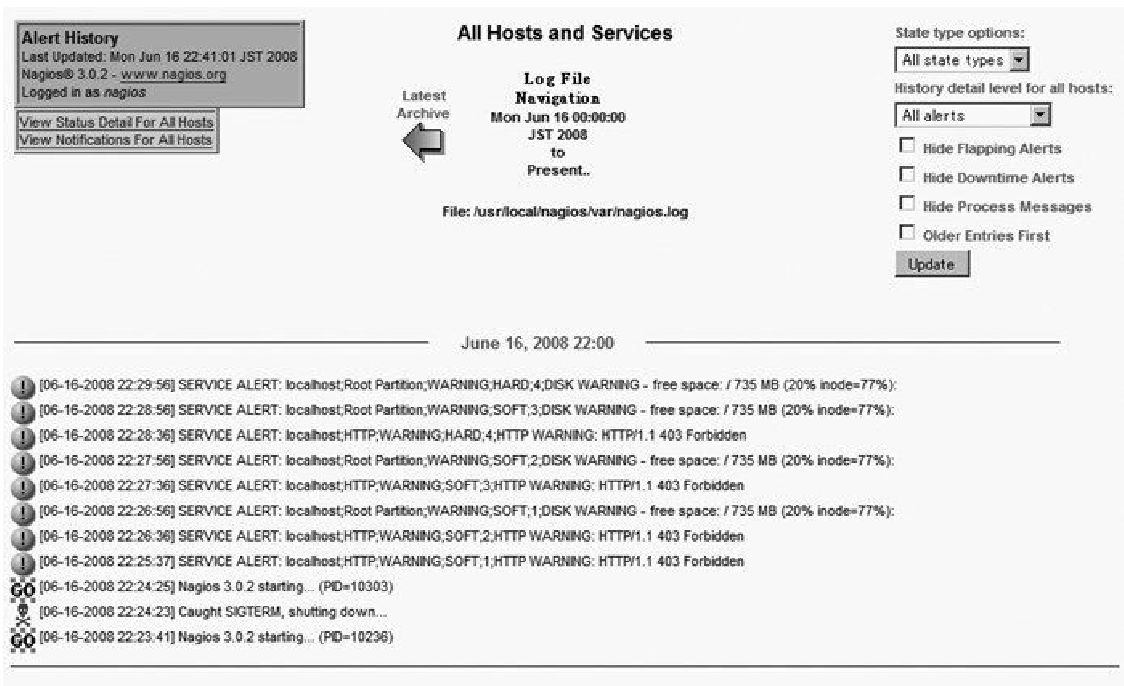


图 **5.1.4　Nagios** 的 **Alert History** 界面

5.1.6　Nagios 的基本使用方法

鉴于 Nagios 的配置极具弹性化，可能不太容易上手。这里我们以网络服务器的监控为例，介绍一下 Nagios 的基本使用方法。

主机和服务的定义

首先，登录正在运作服务的主机（代码清单 5.1.5）。其次，针对各个服务在主机组中进行分配，对每个主机组进行监控。

增加的服务有两种：整个服务器上共通的服务，以及与各个服务器的工作相应的服务。监控服务器中共有的服务时，可以使用 `check_ping` 命令监控主机的运行状态，通过`check_snmp` 命令确认磁盘的剩余空间，这是基础的 SNMP 监控服务。

Apache 之类的 Web 服务器的情况下，可以使用`check_http`；MySQL 服务器的情况下，可以使用`check_mysql` 命令。Official Nagios Plugins 中包含了`check_mysql` 命令。

代码清单 **5.1.5**　主机与服务的配置案例

```
define command{
    command_name check_mysql
    command_line $USER1$/check_mysql -H $HSTADDRESS$ -u $ARG1$ -p
$ARG2$ -P $ARG3$
}

define host{
   use         linux-server
   host_name   databaseserver1
   alias       databaseserver1
   address     192.168.0.100
}

define hostgroup {
   hostgroup_name database-servers
   alias Database Servers
   members databaseserver1
}

define service{
   use http-service
   hostgroup_name database-servers
   service_description MySQL
   check_command check_mysql!nagios!nagios!3306
}
```

通知

根据运行监控，当发现异常或警告时，尽可能迅速地通知管理者进行应对是非常必要的。作为实现这项工作的基本功能，需要发送异常和警告内容的邮件。因为这里可以启动任意命令，所以根据所安装的命令情况，有时还需要向 IRC、IM 等发送通知。

举例来说，假设笔者要把故障通知发到各个运维人员的手机上，还要给 IRC 发送通知。IRC 主要被用于共享应对故障时的信息，通过向 IRC 中发送 Nagios 的通知信息，可以很好地把握发生复杂故障时的状况。

- ——邮件

  邮件是发送 Nagios 故障通知的基本方式。故障一旦发生，如图 5.1.5 所示的邮件就会被发送。默认的配置是主机的情况下使用 `host-notify-by-email` 命令，服务环境的情况下使用 `notify-by-email` 命令。各种命令的配置实例请参阅代码清单 5.1.6。

  图 **5.1.5　Nagios** 的通知邮件示例

```
From: nagios@example.com
Subject: CRITICAL 1011/http_service
To: maintenance@example.com

***** Nagios  *****

Notification Type: PROBLEM

Service: http_service
Host: 1011
Address: 192.168.1.11
State: CRITICAL

Date/Time: 02-08-2008 12:37:51

Additional Info:

(Service Check Timed Out)
```

代码清单 **5.1.6**　邮件通知命令的配置

```
# 'host-notify-by-email' command definition
define command{
  command_name    host-notify-by-email
  command_line    /usr/bin/printf "%b" "*****Nagios *****\n\
nNotification Type: $NOTIFICATIONTYPE$\nHost: $HOSTNAME$\nState:
$HOSTSTATE$\nAddress: $HOSTADDRESS$\nInfo: $HOSTOUTPUT$\n\nDate/
Time: $LONGDATETIME$\n" | /bin/mail -s "$HOSTSTATE$ $HOSTNAME$!"
$CONTACTEMAIL$
        }

# 'notify-by-email' command definition
define command{
  command_name    notify-by-email
  command_line    /usr/bin/printf "%b" "***** Nagios *****\n\
nNotification Type: $NOTIFICATIONTYPE$\n\nService: $SERVICEDESC$\
nHost: $HOSTALIAS$\nAddress: $HOSTADDRESS$\nState: $SERVICESTATE$\
n\nDate/Time: $LONGDATETIME$\n\nAdditional Info:\n\
n$SERVICEOUTPUT$" | /bin/mail -s
" $SERVICESTATE$ $HOSTALIAS$/$SERVICEDESC$ " $CONTACTEMAIL$
  }
```

- **——IRC**

在 Hatena 网站的架构中，除邮件的方式外，还可以向 IRC 的相应频道发出警告。当故障发生的时候，管理员需要在考虑应对方法的同时，与多方联络进行沟通。在这种情况下，随时掌握不断变化的服务器的状态并共享信息，这样才可以有效地应对故障。而且，当发生影响范围大的故障时，使用邮件的方式就会造成大量邮件的发送，因此邮件监控方式的直观性可能会受到影响。此时，使用 IRC 的方式，通过对 IRC 的频道日志进行确认，就能够完全掌握全部状况。

在使用 IRC 的方式传递信息时，需要准备 IRC 服务器、IRC 消息机器人、向消息机器人传递信息的客户端这三者。IRC 服务器使用了标准的 ircd；IRC 机器人使用名为 Kwiki::Notify::irc 的 CPAN 模块，在该模块中包含名为“notify-irc.pl”的消息机器人；客户端使用了将同一模块的 Kwiki::Notify::IRC.pm 作为单一脚本来运行，这里也使用到了“notify_irc.pl”（代码清单 5.1.7、代码清单 5.1.8）

代码清单 **5.1.7**　使用 **notify_irc.pl** 所需的设定

```
define contact{
  contact_name irc
  alias irc-bot
  email test@example.com
  service_notification_period 24x7
  host_notification_period 24x7
  service_notification_options w,u,c,r
  host_notification_options d,u,r
  service_notification_commands notify-irc
  host_notification_commands host-notify-irc
}

# 'notify-by-irc' command definition
define command{
  command_name    notify-irc
  command_line    $USER1$/notify_irc.pl "$SERVICESTATE$ $HOSTALIAS$
/$SERVICEDESC$($HOSTNAME$)"
}
# 'host-notify-by-irc' command definition
define command{
  command_name    host-notify-irc
  command_line $USER1$/notify_irc.pl "$SERVICESTATE$ $HOSTALIAS$($HOST
}
```

代码清单 **5.1.8**　**notify_irc.pl**

```
#!/usr/bin/perl
use strict;
use warnings;
use POE::Component::IKC::ClientLite;
my $message = shift;

my $remote = POE::Component::IKC::ClientLite::create_ikc_client(
  port => 9999,
  ip => 'localhost',
  name => "Nagios$$",
  timeout => 5,
) or die "Couldn't create IRC connection!";
$remote->post('notify_irc/update', $message);
exit 0;
```

5.1.7　实用的使用方法

本节将要介绍测定可用率及独立插件等 Nagios 的使用方法。

可用率的测定

要测定可用率，首先要能够检测服务的运行。基本做法是，使用服务的全局 IP 地址定义主机，然后再定义作为测定对象的服务。例如，测定 `http://www.hatena.ne.jp/` 的可用率的 Nagios 配置就如代码清单 5.1.9 所示。`check_vhost` 命令可以通过使用`check_http` 命令，在 FQDN (Fully Qualified Domain Name，完全限定域名) 中指定相应域名，从而来指定任意服务器。

接下来，将可用率记录为图表化的形式。代码清单 5.1.10 是 Hatena 的分析脚本，是在 Nagios 的 Web 管理界面中截取的。该脚本每日运行一次，这样服务的可用率就可以以明了的图表形式向外部展示。图 5.1.6 中展示了图表化的可用率。

代码清单 **5.1.9**　测定可用率的配置

```
define service {
  use generic-service
  host_name hatena-www.hatena.ne.jp
  service_description hatena-www
  check_command check_vhost!www.hatena.ne.jp!/
}

define host {
  use generic-host
  host_name hatena-www.hatena.ne.jp
  address 59.106.108.86
  alias hatena-question
}

# 'check_vhost' command definition
define command{
  command_name    check_vhost
  command_line    $USER1$/check_http -H $ARG1$ -u $ARG2$ -t 120
}
```

代码清单 **5.1.10**　将可用率图表化的脚本

```
#!/usr/bin/env ruby
require 'rubygems'
require 'hatena/api/graph'
require 'hpricot'
require 'pathname'
root = Pathname.new(__FILE__).parent
require root.parent.join('lib/hatena_consts')

targets = [
          { :host => "hatena-a.hatena.ne.jp",
            :service => "hatena-antenna",
            :id => "hatenaantenna",
            :graphname => 'availability' },
]

targets.each do |target|
  cmd = "curl 'http://192.168.0.1/nagios/cgi-bin/avail.cgi?host=#{t
arget[:host]}&service=#{target[:service]}&assumeinitialstates=yes&
assumestateretention=yes&assumestatesduringnotrunning=yes&includes
oftstates=no&initialassumedhoststate=0&initialassumedservicestate=
0&timeperiod=last31days'"
  body = `#{cmd}`

  graphname = target[:graphname]
  count = Hpricot(body).search('td.serviceOK').last.innerText.to_f

  g = Hatena::API::Graph.new(target[:id], HatenaConsts::HATENA_PASSWD)
  puts "#{target[:host]}/#{target[:service]} #{target[:id]},
#{graphname}, #{count}"
  g.post_data(graphname, :value => count)
end
```

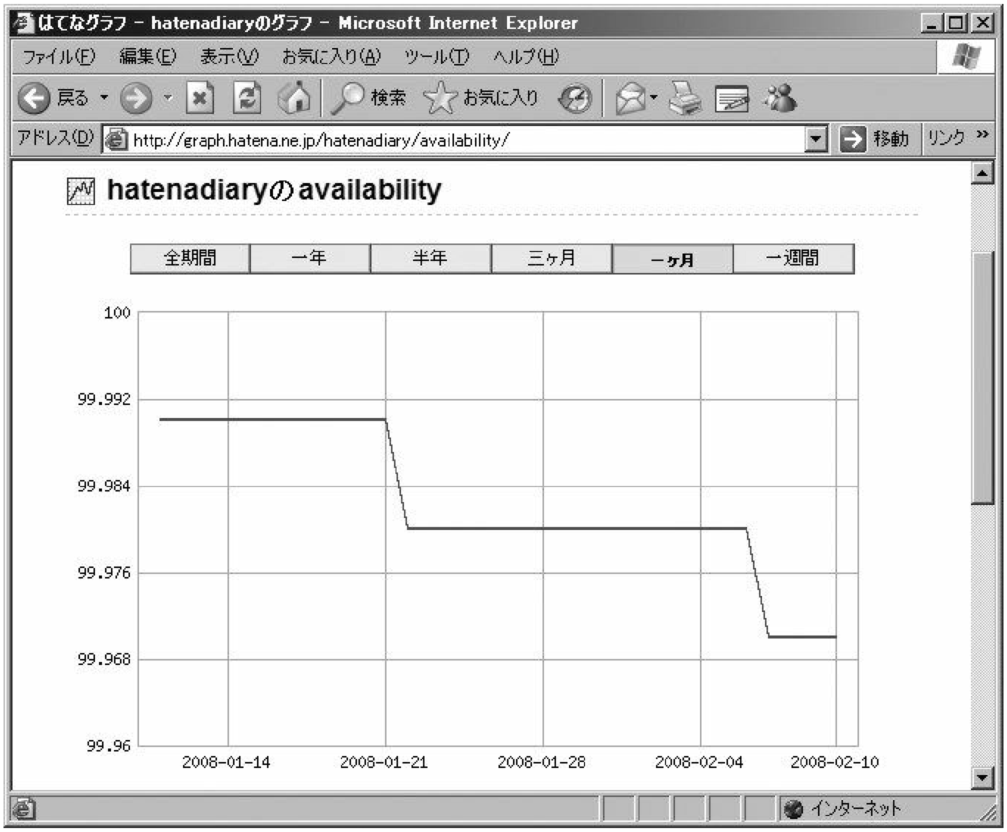


图 **5.1.6** 可用率图表（使用 **Hatena Graph** 制作）

独立插件

Nagios 中具备了很多监控用的命令，但却无法监控未响应的服务状态或特殊的硬件状态等。这种情况下有三种方法：使用 NRPE（Nagios Remote Plugin Executor）、使用 SNMP、使用独立插件。

使用 NRPE 可以便捷地加入监控对象，但是需要在每个服务器中给 Nagios 确立新的进程，因此 Hatena 网站中就没有使用这个办法。而是在像 MySQL 那样可以进行远程访问的情况下，使用独立插件进行监控；反之当远程主机不能访问时，则通过 net-snmp 经由 SNMP 进行监控。

下面介绍三个独立插件的示例，即“MySQL 同步监控”“MySQL 进程监控”“memcached 监控”[5] 。

> [5] 关于这三个独立插件 check_mysqlrep.sh、check_mysql_process.sh、check_memcached.sh，请参考本书最后的补充信息。

- **——MySQL 同步监控……check_mysqlrep.sh**

监控 MySQL 同步是否正确运行（代码清单 5.1.11、代码清单 5.1.12）。该脚本对 MySQL 服务使用了`show slave status` 命令，来监控复制操作是否正常。一旦发生错误停止，就会发出相应的错误通知。

代码清单 **5.1.11**　部署 **check_mysqlrep.sh** 所需的配置

```
# 'check_mysqlrep'
define command{
  command_name check_mysqlrep
  command_line $USER1$/check_mysqlrep.sh -H $HOSTADDRESS$ -u $ARG1$ -p
}
```

代码清单 **5.1.12**　**check_mysqlrep.sh**（摘录）

```
status=`$mysqlpath/mysql -u $user -p$pass -P $port -h $host -e
'show slave status\G' 2>&1`
if [ $? -gt 0 ]
then
  echo $status | head -1
  exit $STATE_CRITICAL
fi

badcount=`echo $status | grep Running | grep No | wc -l`;
lasterror=`echo $status | grep Last_error`
IFS=$_IFS_OLD

if [ $badcount -gt 0 ]; then
  echo "NG: $lasterror"
  exit $STATE_CRITICAL
else
  echo 'OK'
  exit $STATE_OK
fi
exit $STATE_UNKNOWN
```

- **——MySQL 进程数监控……check_mysql_process.sh**

对 MySQL 正在处理的查询数（进程数）进行监控的脚本（代码清单 5.1.13）。在这个脚本中，通过向 MySQL 服务器发出 `show processlist` 命令，查看 MySQL 服务器正在处理的进程数，另外在查看进程数的时候也算进了 sleep 状态的进程。

代码清单 **5.1.13　check_mysql_process.sh**（摘录）

```
processcount=`$mysqlpath/mysql -u $user -p$pass -P $port -h $host
-BNe 'show processlist' | awk '{print $5}' | grep -v Sleep | wc -l`;

echo "processcount:  $processcount"
if [ $processcount -ge $crit ]; then
  exit $STATE_CRITICAL
elif [ $processcount -ge $warn ]; then
  exit $STATE_WARNING
else
  exit $STATE_OK
fi
exit $STATE_UNKNOWN
```

- **——memcached 的监控……check_memcached**

监控在内存中运行的缓冲工具 memcached 是否正常运行（代码清单 5.1.14）。该脚本发布在 Ian Zilbo 的主页 [6] 上。在该脚本中，连接指定的 memcached 服务器，来确认是否可以正常地设定或读取数值。

代码清单 **5.1.14　check_memcached**（摘录）

```
my  $memd = new Cache::Memcached {
    'servers' => [ "$host:$port"  ],
    'debug' => 0,
    'compress_threshold' => 10_000,
};

unless ( $memd->set( $key , "Nagios Check key", 4*60 ) ) {
  print "unable to set memcached $key";
  exit $ERRORS{'CRITICAL'};
```

```
}

my $val = $memd->get( $key );
if ( defined($val) and $val eq "Nagios Check key" ) {
    exit $ERRORS{'OK'};
}
else {
  print "unable to get memcached $key/wrong value returned";
  exit $ERRORS{'CRITICAL'};
}
```

[6] ***URL*** http://zibo.com/ （本书执笔时该网站已经无法打开）。

5.1.8　小结

在服务器监控这种需要稳定运行的领域，使用 Nagios 是很明智的选择。而且 Nagios 非常适合在大规模环境中使用，因为 Nagios 提供了丰富的插件，能够应对各种环境，通过有效使用 Nagios，可以实现基础设施的稳定运行。

5.2　服务器资源的监控 ——Ganglia

5.2.1　服务器资源的监控

本节将介绍服务器资源监控的相关内容。前半部分分析监控的作用等，后半部分介绍笔者所应用的监控的具体实施策略。

监控的目的

首先对监控这种行为进行分析整理。监控的目的用一句话来说就是观察服务器行为的变动。说到观察行为的变动，就是指持续记录下面这些表示服务器状态的指标：

- **CPU** 使用率
- 内存使用率

- **load average**
- 网络流量

并将其可视化，以便更清楚地把握变化的具体情况及倾向。

与需要及时处理的服务监控（也可以说检测异常）不同，观察行为的变化看似用处不大，但是在后面的实践中就会体会到其价值。

此外，诸如在电子杂志、电视商业广告、大规模的门户网站刊登了某站点的广告的情况下，基于广告的宣传效果，该站点可能会瞬间产生大量的访问，这时，直到访问稳定下来之前，管理者一定会忙得不可开交。这样一来，一段时间之后谁还会知道系统的瓶颈究竟在哪里呢？而如果能将过去服务器资源的变化用图表呈现出来，就可以清楚地进行比较，从而也更容易分析。

另外，在环境没有变化但是观测的数值出现波动的情况下，就可以预测可能会发生故障。举例来说，在 I/O 进程的等待数持续增加居高不下的情况下，就可能是磁盘故障导致 I/O 出现了性能瓶颈。通过使用监控，可以及时应对上述状况。

5.2.2 检测工具的讨论

那么，实际上监控是怎样部署的呢？虽说从零开始建立相关机制也是可行的，但目前已经存在了很多监控相关的工具，可以帮助我们收集资料并进行图表化，因此我们可以直接使用这些现成的工具来部署。下面是几个比较常用的工具：

- **Munin** ***URL*** **http://munin.projects.linpro.no/**
- **Cacti** ***URL*** **http://www.cacti.net/**
- **Centreon**（旧称 **Oreon**）***URL*** **http://www.centreon.com/**
- **Monitorix** ***URL*** **http://www.monitorix.org/**
- **NetMRG** ***URL*** **http://www.netmrg.net/**

- **collectd *URL* http://collectd.org/**

这里讲一下笔者使用 Munin 和 Cacti 的感受。因为 Munin 中存在非常多的插件，所以即便不编写程序或进行详细的设定，也可以使多种资源报表图表化。Cacti 的优点在于清晰地划分了层（收集资料、定义资料、绘制图表）的结构，以及所有的配置都可以在浏览器上运行。

一般情况下，在增加或者删除作为监控对象的节点（服务器）的时候，上述两种工具都必须进行改写配置的操作，而且图表的一览性也会变差，因此不适合用于监控大量服务器。

5.2.3 Ganglia ……面向大量节点的图表化工具

那么在服务器集群中究竟要使用什么工具呢？这时就该 **Ganglia** [7] 出场了。根据 Ganglia 的官网介绍，最初 Ganglia 是以在组或分布式计算这些有大量节点的环境中使用为前提而创建的监控工具。以下列举几点在实际使用中比较便利的地方。

> [7] ***URL*** http://ganglia.info/

首先第一点，增加、删除节点（服务器）时无需变更配置。在增加的节点中只需修改代理（Gmond）就可以了。这样一来，在与采集数据并进行图表化作业的程序（Gmetad）通信时，就可以将图表化对象增加进相应的组了。然后活用组播通信，就没必要配置 IP 地址了，也不用为了检测出节点而在整个子网的范围内进行 SNMP 了。

第二点是图表的一览性强。请看图 5.2.1。像这样将所有节点的图表都以网格的方式表现出来，更易于进行宏观的比较。

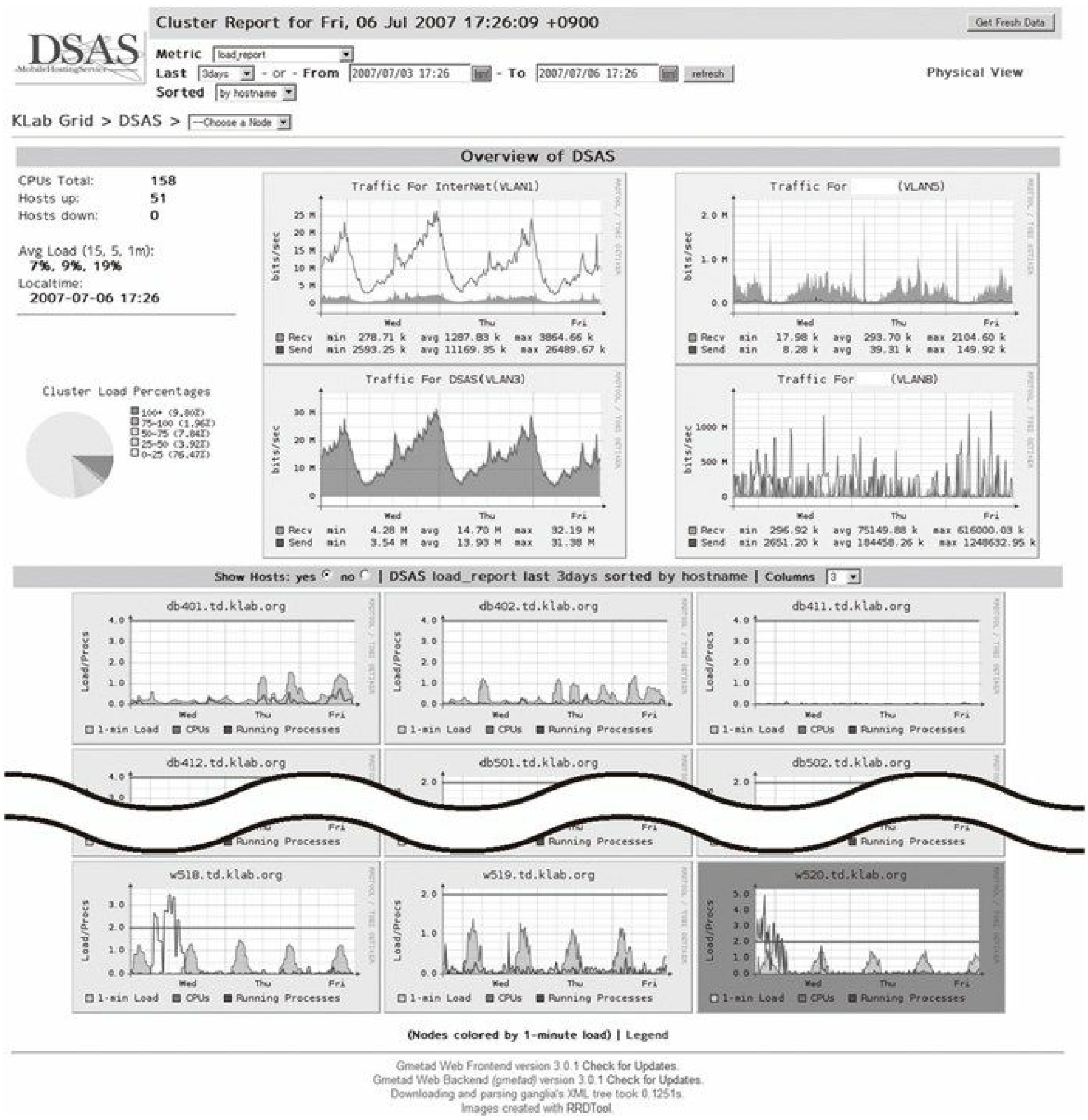


图 **5.2.1　Ganglia**

相反，也列举几点不便之处。

首先，图表的种类没有那么多。只能制作 CPU 和内存使用率、流量之类的基本图表，种类没有 Munin 多。另外在增加单独的图表的情况下，如果数值是一个种类的话，只需运用 gmetric 命令来发送数值，就可以简单地制作图表。但如果在一个图表中描述多个数值的话（例如在一个图表中对 I/O 读取和 I/O 写入两方面进行描述），就需要以 Ganglie 自身的代码（PHP）来实现，因此较为麻烦。

第二点，图表中指定显示时间的选择项是固定的，只有一小时、一天、

一周、一个月、一年，除此之外都不能选择。比如“有大量访问的那一天几点到几点的图表”就不能显示，也就是说，不能灵活地显示某个时间段的状况。一般来说，因为所有时间段的观测资料都被保存了下来，所以只需对指定显示期间的用户接口和描述逻辑稍加设定，就应该能获得任意时间的图表。笔者使用了 Yahoo! UI Library[8] 的日历进行了上述改造，这样就能够方便地调取所需日期的信息了。

[8] ***URL*** http://developer.yahoo.com/yui/

*　*　*

虽然直接使用 Ganglia 还存在一些小问题，但是在有大量节点的环境中使用时，因为基本的设计和相关机制已经比较完善，而且该工具又是使用 PHP 编写的，所以整体的可读性及定制性非常强大。因此使用 Ganglia 时，可以根据自己的需要做出相应的改进。接下来，我们将介绍增加图表的方法。另外顺便一提，本节中使用的 Ganglia 的版本是 3.0.5。

5.2.4　将 **Apache** 的进程状态图表化

作为向 Ganglia 中增加自定义图表的示例，下面介绍一下将 Apache 的进程状态图表化的方法。

若开启 Apache 所附带的 mod_status 模块，就可以立即了解到各 httpd 进程的工作状态（表 5.2.1）。例如，将 httpd.conf 按照代码清单 5.2.1 设置的话，就能够通过访问“`http://example.org/server-status`”来查看当前的状态信息。而且访问 `/server-status?auto` 的话，就 会返回便于程序处理的响应形式。

表 **5.2.1**　**Apache** 的状态

标记	意义
_	等待连接中
S	正在启动

R	正在读取请求
W	正在发送响应
K	应Keep-Alive的要求待机中（保持常连接进程，可发送多个文件）
D	正在请求DNS
C	正在关闭连接
L	正在写入日志
G	处理完成（Graceful）
I	空闲进程（整理Idle的worker）
.	没有当前进程

代码清单 **5.2.1　mod_status** 的设定

```
<Location /server-status>
  SetHandler server-status

  Order Deny,Allow
  Deny  from all
  Allow from 192.168.31.0/24
  Allow from 127.0.0.1
</Location>
```

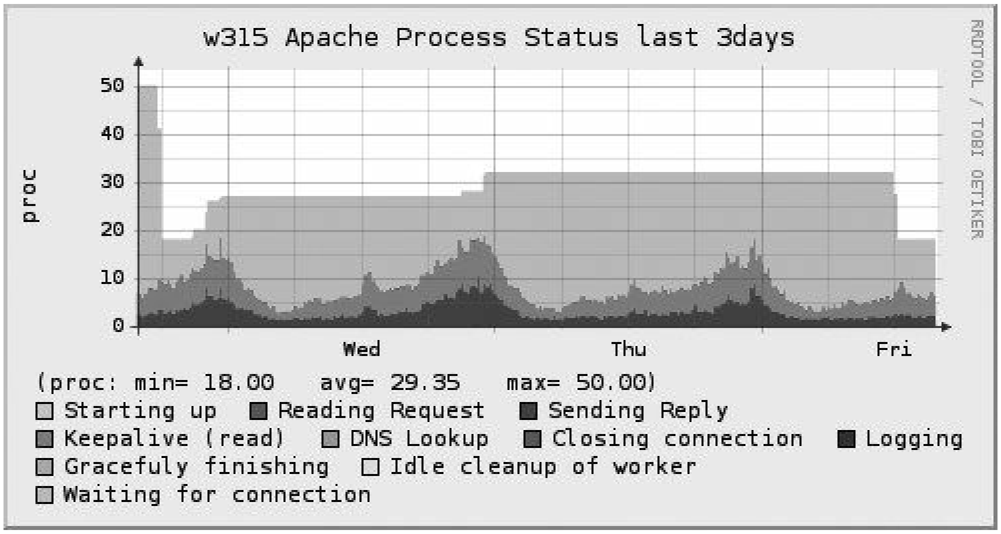


图 **5.2.2　Apache** 进程状态的图表

在图 5.2.2 中，每个进程的状态用不同的颜色进行了区分。通过制作这样的图表，比起简单地表述为“服务器繁忙”，可以清楚地看出具体是由 Keep-Alive 超时等待所造成的，还是由日志写入的 I/O 等待所造成的，一目了然。

在 **Ganglia** 中增加图表的方法

在 Ganglia 中增加图表时，如果要用一个图表描述一个监控值，则只需使用`gmetric` 命令即可。`gmetric` 将使用组播来播报监控值等信息，gmetad 则会将监控值存储下来。

另外，将多个监控值放在一个图表中进行描述的情况下，则需要着手修改 Ganglia 的源码。因为各个监控值是保存在不同的图表文件中的，所以要想办法来统一读取这些监控值，并将其描述在一个图表中。

尝试增加多个图表

以下将实际尝试增加 Apache 的进程状态图表。首先进行代码清单 5.2.1 的配置，然后访问“`http://localhost/server-status?Auto`”，来确认是否能得到应答。

一旦得到确认，就会访问该 URL 并处理其应答，执行使用 gmetric 发送监控值的程序，这里默认每 60 秒获取一次 Apache 的进程信息，并使用了通过 gmetric 发送数据的示例程序 [9] 。

> 9 该示例程序在源代码中可以查看（apache-status）。本书的源代码可以通过以下链接下载：
> ***URL*** http://www.ituring.com.cn/book/download/b70de735-d478-466b-96ac-6782db82f642

至此，在 Ganglia 的 Web 界面的集群视图的 Metric 下拉菜单和主机页面视图下，应该会显示个别监控值的图表（ap-closing 等以 ap- 开头）。

接下来，对监控值进行归纳总结，输出一个如图 5.2.2 所示的图表。这里对变更的文件进行简单的说明。全部的变更内容请参考源代码中的 ganglia.patch。

- **conf.php、my-conf.php**

  在 conf.php 中引用（include）一个 my-conf.php 文件，有关本次变更的配置项目在 my-conf.php 中修改

- **functions.php**

  增加 run_apache 函数，通过将相关参数传递到主机，并根据传递的 boolean 值来决定是否生成 Apache 图表。这里只是单纯地通过主机名进行判断

- **graph.php**

  变更较多行的是 graph.php，但是并不复杂。这里只是用 RRDtool 语法发出了生成图表的指示，因为读取的数据较多，所以行数也随之增加

- **templates/default/host_view.tpl、host_view.php**

  定制主机视图。在模版中通过“functional”标签增加插入点，在 host_view.php 中，当 run_apache 为真时，就将“functional”标签分配到显示 Apache 图表的 HTML 文件中

- **header.php**

  在集群视图的 Metric 下拉菜单中显示 Apache 的图表（Apache_Proc_report）

其他的自定义图表

除了上述列举的图表之外，还有其他一些有用的图表。和之前的 Apache 进程状态的图表类似，只需自定义 Ganglia 就可以生成，大家不妨尝试一下。

- **MySQL** 的各种缓存（查询缓存和密钥缓存等）命中率的图表（图 **5.2.3** ❶）
- **MySQL** 每秒处理的查询数的图表（图 **5.2.3** ❷）
- **MySQL** 的 **SELECT**、**INSERT**、**UPDATE**、**DELETE** 查询比例的图表（图 **5.2.3** ❸）
- **MySQL** 的 **lnnoDB** 的表空间剩余量的图表（图 **5.2.3** ❹）
- **MySQL** 连接数的图表（图 **5.2.3** ❺）
- **Tomcat** 的堆内存使用状况的图表（图 **5.2.3** ❻）

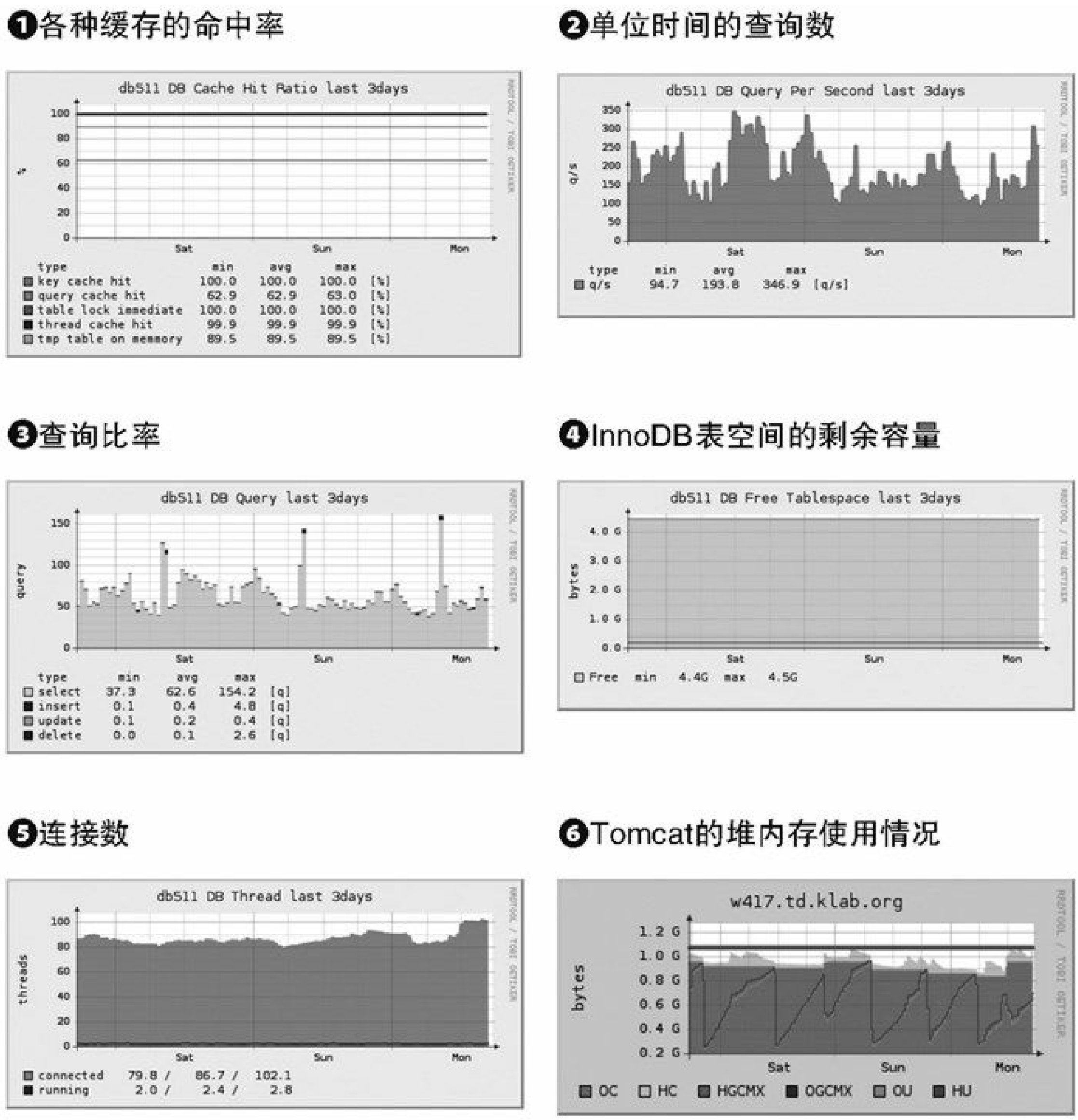


图 **5.2.3**　自定义图表

5.3　高效的服务器管理 **——Puppet**

5.3.1　实现高效的服务器管理的工具 **Puppet**

由于现行的服务正逐步趋向于大规模化，随之而来的是服务器台数和服务器管理费用的增加。而如果将提供个人服务的一台到数台不等的服务器的管理方法应用到企业的数百台到数千台不等的服务器的管理上，就需要采取适当的手段才能完成。为此，对于已成规模的企业服务器管理

来说，不仅需要高效、准确地维持平衡的运行环境，而且还可能需要将配置的变更想办法同步到所有服务器上。

Puppet [10] 是一个能够实现高效地管理服务器的工具，并且适用于大规模的运行环境。通过使用Puppet，原本需要手动登录各个服务器进行的操作现在基本上都可以自动完成了。例如：

> [10] ***URL*** http://puppet.reductivelabs.com/

- 投入新型服务器
- 变更已有服务器的配置

通过使用 Puppet，以上两种工作将变得更加轻松。而且还能避免因人为疏忽而忘记进行某些配置。

本节中所介绍的内容以 CentOS 5.0 的 Puppet 0.24.4 版本为准。

5.3.2　Puppet 的概要

Puppet 是由 Reductive Labs 开发，使用 Ruby 语言描述的开源服务器自动化管理工具。

在 Puppet 的各服务器 Manifest（配置文件）中，各个 Manifest 通过类进行定义，而且定义新的 Manifest 时还可以继承已经定义好的旧的 Manifest 等，因此其具有面向对象的灵活性的特性。Puppet 于 2003 年开始开发，最近一两年得到了越来越多的关注，国内也有一些大企业引入了 Puppet。

Puppet（图 5.3.1）通过在各个服务器上运行的 **puppetd** 及管理服务器上运行的 **puppetmasterd** 这两个守护程序来进行运作。每个服务器的 puppetd 会定期（默认 30 分钟）向 puppetmasterd 发出询问，把得出的结论与现状进行对比，以更新相应的服务器配置。此时，配置文件可以从 puppetmasterd 下载。当然，也可以不进行定期的询问，而是直接运行 `puppetd` 命令，以确认并更新相应的服务器配置。另外，通过在 puppetmaster 服务器上执行`puppeturn` 命令，也可以更新相应的服务器配置。

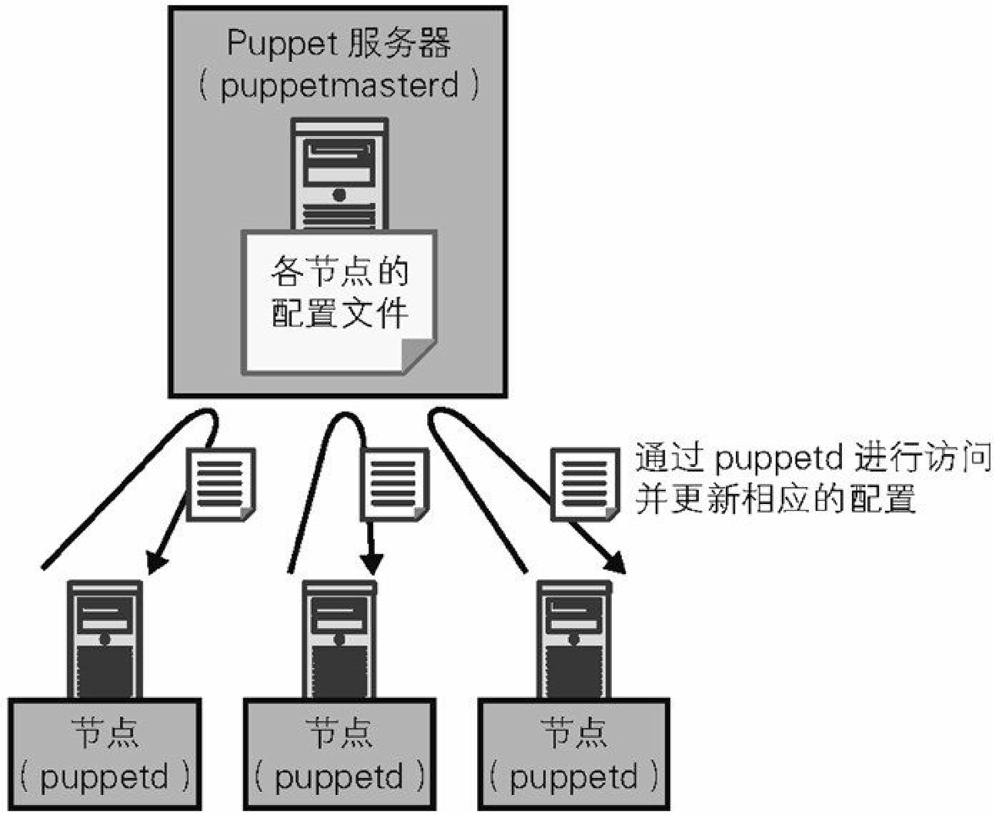


图 **5.3.1**　使用 **Puppet** 管理服务器

puppetd 和 puppetmasterd 之间的通信还采用了 SSL 加密设计以确保安全。

5.3.3　Puppet 的配置

本节将以 Apache 的 Web 服务器的设定为例，来讲述 Puppet 配置的概要。Puppet 的配置可分为以下两个部分：

- **Puppet** 自身的配置（**/etc/puppet/puppet.conf**）
- 部署由 **Puppet** 设定的服务器配置内容的配置文件（**Manifest**）

这里对 Manifest 的内容进行进一步说明。在 Puppet 的 Manifest 中，节点（作为 puppetd 的配置对象的服务器）中会被分配对各服务器的配置进行定义的类。此外，类还能像面向对象的语言那样继承其他类。该配置通常保存在 /etc/puppet/manifests/ 中。若将全部的配置都集中在一个文件进行描述的话，就会导致文件变得很大，所以建议根据服务器的功

能等进行有效的划分。

节点的定义

首先，对代表各个服务器的节点的配置内容进行定义。这里并不直接进行具体的配置操作，而是通过指定配置集合的类使其自动完成这些配置操作。一个节点能够指定多个类。代码清单 5.3.1 中就为配置对象的服务器（testserver）通过 Apache 的 mod_perl 模块指定了 Web 服务器的类（apache-mod_perl）。

代码清单 **5.3.1**　节点定义文件（**nodes.pp**）

```
node testserver {
  include apache-mod_perl
}
```

类的定义

类是具体配置的集合。通过继承其他的类，能够统一定义具有类似功能的服务器的相同配置。

代码清单 5.3.2 所显示的 apache-mod_perl 类中定义了各种配置项目，如是否安装了各种 rpm 软件包（httpd、mod_perl），httpd.conf 是否是最新版，httpd 是否已经被启动等。接下来，对具体的配置内容进行说明。

通过代码清单 5.3.2 ❶ 的 package 的声明，可以发现这里指定安装了 httpd、mod_perl、perl-libapreq2 的软件包（`ensure => installed`）。如果还未安装，则会使用包管理系统自动进行安装。

通过代码清单 5.3.2 ❷ 的 configfile 的声明，可以配置 httpd.conf 及 sysconfig/httpd 这两个文件。`require => Package["httpd-mod_perl"]` 语句表明在软件包被安装后，就开始执行文件的分发行为。configfile 并 非 Puppet 中默认配置的声明，而是由后面叙述的 defined 的声明扩充而来的。实际上被分发的文件的地址是由 configfile 声明中的 source 属性指定的。puppetmasterd 同样也兼容文件服务器，

可以将指定的文件分发到目标服务器中。

通过代码清单 5.3.2 ❸ 的 user 的声明，可以对 apache 的用户进行定义。从其官方文档中可以得知，这里是可以配置 password 的，但现行的版本中被注释掉了，因此需要逐台为服务器设定 password。

通过代码清单 5.3.2 ❹ 的 service 的声明，可以发现这里启动了 httpd 进程，并定义了在启动时通过 chkconfig 执行。通过 `subscribe => [File[$path], File[$sysconfigpath], Package['httpd'], Package['mod_perl'] ]` 的描述，可以在安装及更新包，以及更新配置文件时自动重启 httpd 进程。另外通过`enable => true` 的描述，可以在服务器重启时自动启动服务。而在手动进行管理时，往往会忘记将其设定为自动启动。

代码清单 5.3.2 ❺ 的 file 声明中定义了 /ver/www 目录的属性。

代码清单 **5.3.2**　类定义文件（**apache-mod_perl.pp**）

```
class apache-mod_perl {
  package { httpd:   ←❶
    ensure => installed
  }

  package { mod_perl:
    ensure => installed
  }

  package { perl-libapreq2:
    ensure => installed
  }

  $path = '/etc/httpd/conf/httpd.conf'
  configfile { "$path": ←❷
    ource => "/apache-mod_perl/httpd.conf",
    ode => 644,
    equire => Package["httpd-mod_perl"]
  }

  $sysconfigpath = '/etc/sysconfig/httpd'
  configfile { "$sysconfigpath":
    source => "/apache-mod_perl/sysconfig.httpd",
    mode => 644,
    require => Package["httpd-mod_perl"]
```

```
  }

  user { apache: ←❸
    ensure => present,
    uid => 48,
    gid => 48
  }

  service { httpd:  ←❹
    hasrestart => true,
    hasstatus => true,
    ensure => running,
    subscribe => [ File[$path], File[$sysconfigpath], Package['httpd-mod_pe
    enable => true
  }

  file {  ←❺
    "/var/www": owner => apache, group => apache, mode => 755;
  }
}
```

确认配置是否有效

将这些节点定义和类定义文件读入 puppetmasterd，在 testserver 中按以下方式运行 puppetd，testserver 就会启动 Apache 的 mod_per 模块，使其作为 Web 服务器正确运行。假设 puppetmasterd 运行的服务器的 IP 是 192.168.0.1，即：

```
puppetd -o -v --server 192.168.0.1
```

若正确地进行了配置，就会得到以下输出：

```
info: Caching configuration at /var/lib/puppet/localconfig.yaml
notice: Starting configuration run
notice: Finished configuration run in 1.27 seconds
```

5.3.4 配置文件的语法

Puppet 配置文件的语法是 Puppet 独自定义的，大致类似 Ruby 的语法。下面对其进行简单的讲解。在官方文档中还有详细的说明，有兴趣的话可以参考。

资源的定义

配置文件等的资源（Resource）的集合，包含类（Class）、函数（Definition）、节点（Node）三个种类。

- ——类

  类可以定义多个资源的集合。在一个主机上只能创建一个类的实例。下面的 unix 类中定义了 /etc/passed 文件与 /etc/shadow 文件的属性：

```
class unix {
    file {
        "/etc/passwd": owner => root, group => root, mode => 644;
        "/etc/shadow": owner => root, group => root, mode => 440,
    }
}
```

  也可以继承其他类，只修改其中的一部分。例如像下面这样将文件所属组由 root 变更为 wheel：

```
class freebsd inherits unix {
    File["/etc/passwd"] { group => wheel }
    File["/etc/shadow"] { group => wheel }
}
```

- ——函数

  对变量进行操作时可以定义相应的函数。函数与类不同，不可以进行继承操作。另外在一个主机上可以定义多个函数，这也是函数与

类的不同点。

代码清单 5.3.3 中定义了 configfile 函数，设定了一些参数的默认值。

代码清单 **5.3.3　configfile** 函数的定义和默认值的设定

```
define configfile($owner = root, $group = root, $mode = 644,
$source, $backup = false, $recurse = false, $ensure = file) {
  file { $name:
    mode => $mode,
    owner => $owner,
    group => $group,
    backup => $backup,
    recurse => $recurse,
    ensure => $ensure,
    source => "puppet://$server/config$source"
  }
}

$path = '/etc/httpd/conf/httpd.conf'
configfile { "$path":
  source => "/apache-mod_perl/httpd.conf",
  mode => 644,
  require => Package["httpd-mod_perl"]
}
```

- ——节点

节点中定义的内容与类相同，而且能够在真正的服务器上生效。可以使用 hostname 作为主机的标识符，而不使用 IP 地址。以下是将 apache-mod_perl 类的设定同步到 testserver 服务器上：

```
node testserver {
    include apache-mod_perl
}
```

资源

各个类的实体资源可以根据 Type（类型）进行定义。Type 即 file（文件）或 package（包）等具体的设定项目。

- **——file**

  在“file”中可以定义文件的属性。通过定义 sources，可以从 puppetmasterd 上下载文件。另外，通过定义 content，还可以直接写入内容，或者使用模板功能。

  以下指定了 /etc/passwd 文件的所有者与权限：

```
file {
  "/etc/passwd": owner => root, group => root, mode => 644;
}
```

- **——package**

  “package”中定义了包。通过指定`ensure =>installed`，可以在没安装包的情况下，在操作系统上安装这些包。例如在 Red Hat 系统上，通过`yum` 命令安装 rpm 包；在 Debian 系统上，通过`apt-get` 命令安装 deb 包等，可以根据操作系统的不同进行包的安装。

  以下是安装 mysql 包的定义：

```
package { mysql:
  ensure => installed,
}
```

- **——exec**

  在“exec”中可以执行任意的命令。例如在下例中，更新完 iptables 的配置文件后，会重启 iptables。

```
exec { "/etc/init.d/iptables stop && /etc/init.d/iptables start":
  subscribe => File["/etc/sysconfig/iptables"],
}
```

- **——service**

“service”中定义了服务（进程）。具体可以定义服务的启动状态及重启时的行为。

以下代码定义了 httpd 服务的运行（`ensure=>running` ）以及在操作系统启动时自动启动服务（`enable=>true` ）：

```
service { httpd:
  ensure => running,
  enable => true
}
```

对各个服务器的配置进行微调

若要针对各个服务器对设定项目进行微调，可以使用 Puppet 的 facterlibrary 变量，来完成配置的调整工作。例如通过访问 $operatingsystem 变量，可以在 Solaris 和 default 的情况下选择不同的文件配置路径。

```
 path => $operatingsystem ? {
       solaris => "/usr/local/etc/ssh/sshd_config",
       default => "/etc/ssh/sshd_config"
},
```

在命令行上执行`facter` 命令，可以浏览除 $operatingsystem 以外的 各变量的使用方法。

资源间的依赖关系

通过定义资源间的依赖关系，可以指定配置生效的顺序，以及配置生效所需重启的服务。例如在更新 httpd.onf 后，就需要重启 httpd 以使配置

生效。这样做的目的是为了避免配置文件更新后因为没有重启服务而造成配置无效的情况发生。

在下面的例子中，通过在 service 函数中定义 subscribe 变量的值，设定了 /etc/httpd/conf/httpd.conf 配置文件与 httpd 服务的依赖关系。通过这样的配置，在使用 Puppet 时若发生 /etc/httpd/conf/httpd.conf 配置文件更新的情况，就会自动重启 httpd 服务以使更新生效。

```
 $path = "/etc/httpd/conf/httpd.conf"
 configfile { "$path":
   source => "/apache-mod_perl/httpd.conf",
   mode => 644,
 }

 service { httpd:
   hasrestart => true,
   hasstatus => true,
   ensure => running,
   subscribe => [ File[$path] ],
   enable => true
}
```

通过模板完成 **Manifest** 文件的定义

Puppet 的特点就是可以根据用途，使用模板自定义复杂的 Manifest 文件，并同时进行配置的分发操作。在此以 DualMaster（双向同步）MySQL 类与 iptables 类为例进行说明。

- **——DualMasterMySQL** 类

  MySQL 的配置文件（my.cnf），如 server_id 和同步的设定、双向同步的设定等，在不同的服务器中这些配置都是不同的。因此，通过使用模板功能，可以让描述工作变得更简洁。在该类中，通过在各节点的定义中传递参数来调整配置文件。

  首先，在代码清单 5.3.4 中定义 mysql-master-conf 函数；接着，在代码清单 5.3.5 的节点的配置中设定 server_id、master_host 等参数信息；最后，在代码清单 5.3.6 的 multimaster-my.cnf 中定义模板。

进行以上配置后，运行 puppetd，模板就可以根据 server_id 或 master_host 等传递的参数值，生成相应的配置文件，完成实际的服务器的配置。在此需要注意的是，这里之所以不是“server-id”而是“server_id”，是因为 Puppet 的语言规范不允许参数中出现“-”。

代码清单 **5.3.4　mysql-master-conf** 函数

```
define mysql-master-conf($path, $server_id, $master_host = false,
$auto_increment_increment = false , $auto_increment
_offset = false, $log_bin = false, $log_slave_updates = false,
$innodb = false, $replace = true) {
  templatefile { $path:
    source => "mysql/multimaster-my.cnf.erb",
    notify => Service[mysqld],
    replace => $replace
  }
}
```

代码清单 **5.3.5　mysqldb** 节点的定义

```
node mysqldb {
  mysql-master-conf {"my.cnf":
    path => "/etc/my.cnf",
    server_id => "1001",
    master_host => "192.168.1.1",
    auto_increment_increment => "16",
    auto_increment_offset    => "1",
  }
}
```

代码清单 **5.3.6　multimaster-my.cnf**（摘录）

```
server-id    = <%= server_id %>
log-bin

master-host  = <%= master_host %>
master-user  = repli
```

```
<% if auto_increment_increment then -%>
auto_increment_increment = <%= auto_increment_increment %>
<% end -%>
<% if auto_increment_offset then -%>
auto_increment_offset    = <%= auto_increment_offset %>
<% end -%>
```

- **——iptables** 类

在 LVS（负载均衡集群）中根据 DSR（动态路由协议）进行 iptables 设定时，需要根据主机的用途定义不同的 VIP。因此这里每个 VIP 的设定文件都不同，若一个一个地去准备的话就比较繁琐，而通过在 iptables 类中使用模板功能，就可以在定义节点时传递相应的参数，生成恰当的配置文件。

```
node foobar {
  iptables-lvs-conf {"iptables":
    path => "/etc/sysconfig/iptables",
    lvs_iptables => "59.106.108.97:80"
  }
}
```

以上在 iptables-lvs-conf 这个独立的函数中完成了 iptables 的设定。path 变量是文件的配置路径，lvs_iptables 变量是 iptables 中的内容。在这里定义为了“59.106.108.97:80 ”，实际上执行的是：

```
/sbin/iptables -t nat -A PREROUTING -d 59.106.108.97 -p tcp -j REDIREC
```

另外，如果定义为“59.106.108.97:80:81 ”，实际就会执行：

```
/sbin/iptables -t nat -A PREROUTING -d 59.106.108.97 -p tcp -m tcp --d
```

在代码清单 5.3.7 中定义了 iptables-lvs-conf 函数。这里通过 source 变量定义了模板，通过 notify 变量指定了在模板更新时重启 iptables。 templatefile 是通过 functions/utils.pp 定义的函数，会生成相应的模板。

在代码清单 5.3.8 中定义 iptables 类。

configifle 中包含的 iptables_check.sh 是检测是否设定了 iptables 的命令。由于如果在文件配置的路径中指定了原本就不存在的目录就会造成错误，因此这里设置为了 /usr/bin。在 exec 的定义中，通过订阅（Subscribe）configifle 生成的文件，若发生了 sysconfig/iptables 的变更，就会进行重启操作。

最后的 service 定义指定了执行 iptables。为了监控 iptables 服务的状态，使用了（在代码清单 5.3.8 的 ❶ 处）配置好的 iptables_check.sh。

代码清单 5.3.9 是模板文件。根据传递的参数，生成适当的文件。虽说比较理想的方式是将需要传递的参数以数组的形式传递，但这不符合 Puppet 的语言规范 [11]，因此需要将参数断开为多个参数分别进行传输。

代码清单 **5.3.7　iptables-lvs-conf** 函数

```
define iptables-lvs-conf($path, $lvs_iptables = []) {
  templatefile { $path:
    source => "iptables/lvs_iptables.erb",
    notify => Service[iptables]
  }
}
```

代码清单 **5.3.8　iptables** 类

```
class iptables {
  package { iptables:
    ensure => installed
  }
```

```
  $path = '/etc/sysconfig/iptables'
  $binpath = '/usr/local/hatena/bin'

  $checkcmd_path = '/usr/bin/iptables_check.sh'   ←❶
  configfile { "$checkcmd_path":
    owner  => root,
    group  => root,
    mode   => 755,
    source => "/iptables/iptables_check.sh",
  }

  exec { "/etc/init.d/iptables stop && /etc/init.d/iptables start":
    subscribe   => File["/etc/sysconfig/iptables"],
    refreshonly => true,
  }

  service { "iptables":
    status => "$checkcmd_path",
    start  => "/etc/init.d/iptables start",
    ensure => running,
  }
}
```

代码清单 **5.3.9　templates/iptables/lvs_iptables.erb**

```
# Generated by puppet
*nat
:PREROUTING ACCEPT [6180:371400]
:POSTROUTING ACCEPT [42:5009]
:OUTPUT ACCEPT [42:5009]
<% lvs_iptables.split(/,/).each do |lvs_params| -%>
<% lvs = lvs_params.split(/:/) -%>
-A PREROUTING -d <%= lvs[0] %> -p tcp -j REDIRECT<%= lvs.length > 2
? " --dport #{lvs[2]}" : "" %><%= lvs.length > 1 ? " --to-ports
#{lvs[1]}" : "" %>
<% end -%>
COMMIT
# Completed
```

[11] Puppet 的语言规范中规定了传递给模板的参数仅限于符合要求的字符串。

5.3.5 通知操作日志

puppetd 确认服务器的配置后，会根据需要变更（没必要时无需变更）该服务器的配置。这时变更的内容将输出到 syslog。也可以在邮件或日志中进行通知。

- **——tagmail**

  tagmail 的功能是在各服务器中使用 Puppet 时，通过邮件发送日志。例如，在 /etc/puppet/tagmail.conf 中存在如下记录：

  ```
  all: user1@example.com
  apache: user2@example.com
  ```

  all 是通知全部的变更，通过指定 apache 等的标签，可以将与标签相关联的变更进行通知。标签能够在每个类中定义标签名。另外，因为类名默认是标签名，所以还可以对类名进行指定。

- **——puppetmaster.log**

  var/log/puppet/puppetmaster.log 中会输出运行结果。但因为此处的错误信息不是十分准确，所以调试程序时请不要过多使用。

- **——report**

  以 YAML 的形式输出到 /var/lib/puppet/reports 中。每次生效都会生成一个文件，不生效就什么都不会生成。可以利用其他工具进行处理，并生成图表。

- **——puppetd** 中的日志

  像下面这样在各服务器中直接启动 puppetd，也能够运行 puppetd。

  ```
  % sudo /usr/sbin/puppetd --server=192.168.0.1 -o -v --waitforce 60
  ```

因为此时的日志会被详细地输出，因此在调整配置的时候，这种方法就更具参考性。另外，通过使用--noop 参数，可以在不重写配置文件的情况下确认之前进行的设定。

5.3.6 运用

在 Puppet 中，因为大多数的配置文件都能够简单地进行更新，所以配置错误的影响也比较大。例如在 sshd_config 配置错误时，会造成无法登录系统等故障。正因如此，在进行大规模的变更时，采用能够方便地撤销修改的方法就显得尤为重要。

为此可以考虑以下方法：

- 在应用到所有服务器之前，首先在部分服务器上进行测试。平时将 puppetd 的自动更新设为无效，在测试用的服务器中运行 puppetd，如果没有问题，再将 puppetrun 应用到所有的服务器上

- 采用 Subversion 对配置文件进行管理。通过将 /etc/puppet 目录下与 Puppet 相关联的配置文件全部托管在 Subversion 中进行管理，就能实现配置文件的版本管理。据此可以追踪过去的变更记录，在配置不当的情况下还能回滚到之前的配置版本。而且，还可以对 Subversion 不同版本分支的快照记录使用 make 命令，在配置生效前，使用 --noop 参数在 Puppet 中测试这些快照的应用效果，事先检查配置文件的语法是否存在问题

5.3.7 自动配置管理工具的利与弊

从早先的 cfengine 到近期的 Puppet，这些都是著名的开源（OSS）自动配置工具。这些工具虽然看起来非常方便，但在实施时可能会遇到一些问题。

自动配置工具听起来确实是个很不错的选择，但实际使用时却非常麻烦。以下整理了使用该工具时可能会遇到的状况。

- 从本质上来说这并不是什么创新的东西

  自动配置工具只是将原有手动配置的工作自动化，仅此而已。当然人们通常不想改变目前已存的工作流程

- 要记的东西很多，非常麻烦

  在使用自动配置工具配置应用程序的时候，不仅要记住应用程序的配置方法，还必须为应用程序的配置做一些准备工作。常常让人怀疑这么做有价值

- 在某些情况下，可能正是因为使用了自动配置工具才导致了异常的出现

  通常在自动配置工具托管的配置文件中，不能人为地修改这些配置文件，如若修改，就极易导致故障。例如某天你在运行自动配置工具时，无意间覆盖了手动修正的部分，这时不仅系统瘫痪了，自动配置工具也可能没法运行了，这种情况很有可能发生。而且一旦发生，处理起来就会非常棘手。当遇到此类故障时，根本没有充分的时间去调整配置，如果处理故障的人员对这些配置参数理解得不够充分，那匆忙修改可能会埋下隐患

这样看来，自动配置工具的弊端好像太大了，但为了对大量服务器进行高效的管理，自动配置工具就很有存在的必要。例如在变更成百上千台服务器的 sshd_config 的配置文件时，利用配置工具就可以轻松地完成操作，这真是提高效率的法宝啊。

在应用这种自动配置工具时，哪里自动哪里手动是个重要的问题。然而，这个问题并没有什么一般性的标准，通常要根据应用自动配置工具的设备规模和管理人员的干劲来决定。

我们的目标是：使用自动配置工具，并维持良好的运行状态。通常比较适合使用自动配置工具的有以下两种情况：

- 设备的台数很多时

- 手动配置容易产生疏漏时

5.4　守护进程的工作管理 ——Daemontools

5.4.1　守护进程的异常终止

通常在系统启动时都会自动启动守护进程（Daemon），但万一在服务器运行过程中守护程序意外终止，那会造成多么严重的事态呢？若 Web 服务器、邮件服务器等服务意外终止的话倒是很容易察觉，但察觉守护进程等的异常就比较困难。

虽然在发生意外情况时，有一些自动重启的安全措施 [12]，但 /etc/init.d 中大多数的启动脚本可能都不具备重启的功能。此外，在对各个启动脚本进行进程监控的同时添加重新启动的处理是比较麻烦的。

> [12] 例如 MySQL 所附带的 mysqld_safe 脚本等。

这样，一个叫作 **daemontools** [13] 的工具就应势而出了。该工具解决了守护进程的运行管理问题。本节中我们就来介绍一下 daemontools（0.76 版本）的使用诀窍。

> [13] ***URL*** http://cr.yp.to/daemontools.html 和 daemontools 类似的工具还有 runit（http://smarden.org/runit/ ）。

5.4.2　daemontools

daemontools 工具是一个很实用的软件包，它负责管理守护进程的开始、结束、重新启动，以及进程异常终止时的自动启动等工作。

daemontools 是通过若干个程序联合对守护进程进行监控和管理的，具体请参阅图 5.4.1。

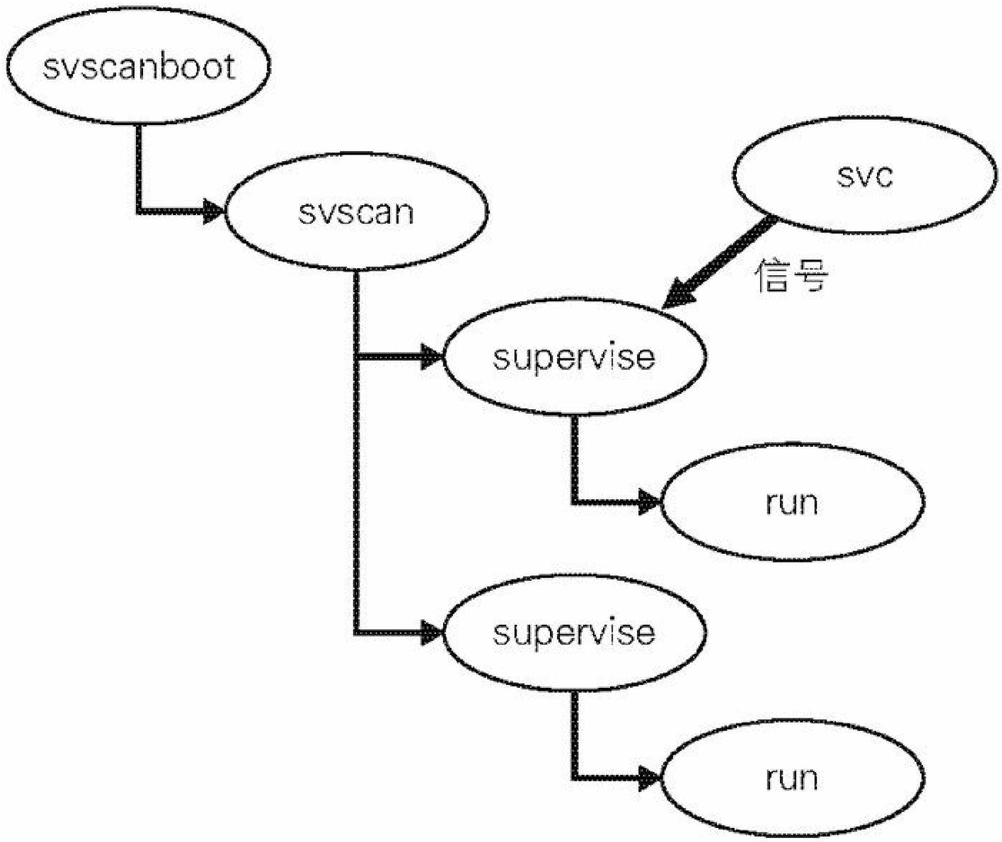


图 **5.4.1　daemontools** 的概要图

首先，svscanboot 启动 svscan。svscan 对指定的目录（默认是 /service）进行监控，在增加了新的守护进程的情况下启动 svscanboot。supervise 运行 run 文件，通过 run 文件启动守护进程。因为 supervise 一次只能对一个守护进程启动，在此由 svscan 对多个 supervise 进行管理。

使用`svc` 命令可以通过 supervise 传送信号到守护进程。

使用 **daemontools** 的原因

使用 daemontools 有以下两个原因：

❶ 在进程终止的情况下，能够快速地自动重启

❷ 能够简单地创建守护进程

关于第一个原因，在守护进程终止的情况下，supervise 能通过监控察觉并迅速地重新启动，这一优势非常突出。此外还可以防止没有注意到守

护进程异常终止的情况出现。

关于第二个原因，通常情况下，要作为守护进程运行，需要进行各种处理 [14] ，例如：

> [14] 在某些操作系统中，也存在集中进行这些处理的 daemon(3) 函数。

- 从控制端断开
- 将当前目录变更为根目录（/）
- 将标准输入输出重定向到 **/dev/null**（或者其他文件）

稍微有些棘手。

但如果使用 daemontools，只要满足某个条件（详情后述），任何程序都可以作为守护进程。赶快试试自己编写监控脚本来创建守护进 程吧。

成为守护进程的条件 **……**在前台运行

在上节中提到了使用 daemontools 创建守护进程需要满足某个条件，这个条件就是"在前台（Foreground）运行"。

httpd、sshd 这些一般的守护进程，由于是在后台进行 folk 操作的，因此不在 daemontools 的管理下。守护进程中提供了在前台运行的选项（例如 sshd 中的`-D` 选项），否则就需要使用 daemontools 自带的 `fghack` 工具了。

在自行编写守护进程的情况下，只需使用 while 或 for 进行无限循环，在程序不终止的前提下，在前台持续运行。

另外，daemontools 中还附带了优秀的日志收集工具`multilog` 。但要使用`multilog` ，需要将守护进程的标准输出（或标准错误输出）输出到日志中。

5.4.3　守护进程的管理方法

下面介绍守护进程的创建、终止和重启的操作方法。

创建守护进程

此处将需要创建的守护进程命名为“xxxd”。

首先，请先创建一个放置这些守护进程文件的目录。使用这一目录来整理有关守护进程的文件，可以方便后期管理。具体操作方法如下：

- **/etc/daemon** → 放置守护进程的子目录
- **/etc/daemon/xxxd** → **xxxd** 用的文件夹

接下来，创建 supervise 运行的 /etc/daemon/xxxd/run 文件。这是一个典型的 run 文件 [15]，run 文件通过 shell 脚本进行有效用户的变更及环境变量的配置后，就可针对程序使用 exec 命令启动相应的守护进程。

> [15] 在以下页面中可查看 run 文件的示例脚本，请参考。
> ***URL*** http://smarden.org/runit/runscripts.html

代码清单 5.4.1 中展示了示例代码，以下进行一些简单的说明。

❶ 将标准错误输出重定向到标准输出上

❷ 运用后面的指令替换掉该进程

❸ 变更有效用户

❹ 重设环境变量，配置必要的环境变量

❺ 若 **env** 子目录中存在文件，则据此设定环境变量

❻ 执行守护进程的程序

若需要使用`multilog` 工具来收集日志，则可在该目录下创建下一级子目录 log 和文件 log/run（图 5.4.2、代码清单 5.4.2）。

代码清单 **5.4.1**　典型的 **run** 文件

```
#!/bin/sh
exec 2>&1 ←❶
exec \ ←❷
  setuidgid USERNAME \ ←❸
  env - PATH="/usr/local/bin:$PATH" \ ←❹
  envdir ./env \ ←❺
  /usr/local/bin/xxxd ←❻
```

```
# mkdir /etc/daemon/xxxd/log
# mkdir /etc/daemon/xxxd/log/main
# chown USERNAME /etc/daemon/xxxd/log/main
```

图 **5.4.2**　使用 **multilog** 的情况

代码清单 **5.4.2**　**/etc/daemon/xxxd/log/run**

```
#!/bin/sh
exec setuidgid USERNAME multilog t ./main
```

启动守护进程

在启动守护进程的时候，需要在 svscan 监控的目录（默认为 /service 目录）上，创建指定守护进程用的目录的符号链接，如下所示。这样在五秒钟内就应该会运行 run，并启动守护进程。

```
# ln -s /etc/daemon/xxxd /service/
```

停止、继续、重新启动守护进程

停止、继续、重新启动守护进程的时候，请使用 daemontools 中自带的 svc 命令（表 5.4.1）。

表 **5.4.1**　停止、继续、重新启动

行为	命令	说明
停止	svc -d /service/xxxd	发送TERM信号结束进程，此处并不重新启动
继续	svc -u /service/xxxd	若进程不存在则启动进程；若进程处于停止状态则让进程继续运行
重新启动	svc -t /service/xxxd	发送TERM信号

停用守护进程

若需要停用守护进程，在删除符号链接后，需要使用 svc 命令释放 supervise。

```
# cd /service/xxxd
# rm /service/xxxd
# svc -dx . log

↓此外，以下内容也需要释放
# mv /service/xxxd /service/.xxxd
# svc -dx /service/.xxxd /service/.xxxd/log
# rm /service/.xxxd
```

发送信号

在表 5.4.1 中写明了使用 svc 命令可以发送 TERM 信号，而其他的信号也同样可以发送（表 5.4.2）。

表 **5.4.2**　能够使用 **svc** 命令发送的信号

svc选项	信号
-p	STOP
-c	CONT
-h	HUP
-a	ALRM
-i	INT
-t	TERM
-k	KILL

Keepalived ……run 文件的例子 ❶

这里介绍两个 run 文件的例子。第一个是运用 daemontools 管理 keepalived 时的 run 文件，具体如代码清单 5.4.3 所示。

代码清单 **5.4.3**　**keepalived** 用的 **run** 文件

```
#!/bin/sh
exec 2>&1
exec /usr/local/sbin/keepalived -n -S 1
```

这里 Keepalived 指定了`-n`（`--dont-fork`）选项以在前台运行，在使用 daemontools 管理的情况下可以使用该选项。此外，在这个例子中，

并没有使用 multilog 来记录日志，而是使用了 syslog 的程序模块（Facility）LOCAL1 进行输出。在无盘服务器无法记录日志的情况下，或者想要将日志集中管理的情况下，可以使用 syslog 将日志放到 syslog 服务器中进行记录。

自编的监控脚本 **……run** 文件的例子 ❷

第二个是介绍运用 daemontools 管理自编的监控脚本的例子。代码清单 5.4.4 是 run 文件，和代码清单 5.4.3 的处理是相同的。

代码清单 **5.4.4**　自编的监控脚本使用的 **run** 文件

```
#!/bin/sh
exec 2>&1
exec \
  setuidgid monitor \
  env - PATH="/usr/local/bin:$PATH" \
  envdir ./env \
  /usr/local/bin/monitor-ping
```

接下来是代码清单 5.4.5。monitor-ping 是监控脚本的代码（shell 脚本）。

这里的重点是要进行无限循环（`while true; do`）以确保持续运行，但此处若全力进行无限循环可能会占用主机过多的资源，可以通过 `sleep` 设置一定的间歇。

代码清单 **5.4.5**　监控脚本的代码（**monitor-ping**）

```
#!/bin/sh
[ "$DEBUG"   = '1' ] && TRACE='echo DEBUG:' || TRACE=:
INTERVAL=${INTERVAL:=5}

$TRACE "TARGET_HOSTS: $TARGET_HOSTS"
$TRACE "INTERVAL: $INTERVAL"

alert() {
  host=$1
  # TODO: implement
```

```
  echo "$host is down!!"
}

monitor() {
  host=$1
  if ping -qn -c 1 "$host" >/dev/null 2>&1; then
    $TRACE "OK $host"
  else
    $TRACE "NG $host"
    alert $host
  fi
}

while true; do
  for h in $TARGET_HOSTS; do
    monitor $h
  done
  sleep $INTERVAL
done
```

此外，在代码清单 5.4.5 的脚本中，还引用了脚本以外设定的变量（TARGET_HOSTS、INTERVAL、DEBUG）。因为在 run 文件中使用了`envdir`，所以像图 5.4.3 那样在 env 目录下创建与变量同名的文件，并将文件的内容设为变量的值，环境变量就会反映到 monitor-ping 上。这样一来，即便监控的主机数量有所增加，在不重写 run 文件或监控脚本 monitor-ping 的情况下，也能改变其行为。

```
# echo 'host1 host2 host3' > env/TARGET_HOSTS
# echo 10                  > env/INTERVAL
# svc -t /service/monitor-ping
```

图 **5.4.3**　使用 **envdir** 指定环境变量的方法

5.4.4　daemontools 的实用技巧

最后介绍使用守护进程工具 daemontools 时的技巧：控制服务的启动顺序和常用的 shell 函数。

控制所依赖的服务的启动顺序

在使用 daemontools 管理的守护进程和使用 rc 脚本启动的守护进程之间，有时存在启动顺序的问题。Dns 就是其中一例。

daemontools 的开发者也开发了一个名为 djbdns[16] 的 DNS 服务器。使用 daemontools 可以管理 djbdns，在通过 /etc/inittab 启动 daemontools（的 svscanboot）的情况下，假设运用 rc 脚本启动守护进程，若启动时不能解析 DNS 名称，就可能会造成无法启动或运行异常等情况。这是因为在 daemontools 管理下的 djbdbs 启动前，即进行 DNS 名称解析前，需要确保 rc 脚本的守护进程已经被启动。

> [16] ***URL*** http://cr.yp.to/djbdns.html

要解决这个问题需要下一些工夫。以下介绍使用 KLab 的方法以供大家参考。

请参阅代码清单 5.4.6。首先，编辑 /etc/inittab，启动 svscanboot ；接着，启动 DNS 服务器，记录需要启动的守护进程的启动命令，并编写相应的启动脚本。各脚本启动的服务如表 5.4.3 所示。

代码清单 **5.4.6　/etc/inittab**（摘录）

```
SV:123456:respawn:/command/svscanboot
LG:2345:wait:/etc/init.d/log   >/dev/null
SH:2345:wait:/etc/init.d/share >/dev/null
WE:2345:wait:/etc/init.d/web   >/dev/null
```

表 **5.4.3**　启动的守护进程

启动脚本	启动的服务
/etc/init.d/log	syslog-ng
/etc/init.d/share	网络文件系统（NFS）的客户端

/etc/init.d/web	Web 服务器

另外，在这个启动脚本（/etc/init.d/log）中，实现了类似于代码清单 5.4.7 中的 waitns 的处理。这样在确认解决了 DNS 名称解析的问题后，紧接着就启动了服务用的守护进程。

代码清单 **5.4.7　waitdns**

```
waitdns() {
  while true; do
    dig @127.0.0.1 +short +time=1 {DOMAINNAME} >/dev/null 2>&1 && break
  done
}
```

在代码清单 5.4.6 的启动脚本中有三行 wait[17]，虽然 waitdns 只对第二行 /etc/init.d/log 是必须存在的，但为了保险起见，无论是之后的 /etc/init.d/share 还是 /etc/init.d/web，都还是写上为妙。

> [17] wait 只在操作系统启动时执行一次，作用是等待第 4 项指定的进程执行结束。

常用的 **shell** 函数

以下介绍笔者在实际运用中常用的三个 shell 函数。

- **——daemonup**

  在 /service 中创建符号链接。登录、启动守护进程（代码清单 5.4.8、图 5.4.4）。

  代码清单 **5.4.8　daemonup**

```
daemonup() {
  [ -z "$1" ] && return

  case $1 in
```

```
    */*)
      DAEMONDIR=$1
      ;;
    *)
      DAEMONDIR=/etc/daemon/$1
      ;;
  esac
  [ -d $DAEMONDIR ] || { echo "no such dir: $DAEMONDIR"; return; }

  d=$(basename $DAEMONDIR)
  if [ ! -s "/service/$d" ]; then
    ln -snf ${DAEMONDIR} /service/
  fi
  /command/svc -u /service/$d >/dev/null 2>&1
}
```

```
# daemonup monitor-ping
  （或者）
# daemonup /path/to/monitor-ping
```

图 **5.4.4 daemonup** 的使用示例

- **——daemondown**

与 daemonup 的作用正好相反。停止、删除守护进程，并从 /service 中删除符号链接（代码清单 5.4.9、图 5.4.5）。

代码清单 **5.4.9 daemondown**

```
daemondown() {
  [ -z "$1" ] && return
  if [ -s /service/$1 ]; then
    mv /service/$1 /service/.$1
    /command/svc -dx /service/.$1
    if [ -d /service/.$1/log ]; then
      /command/svc -dx /service/.$1/log
```

```
    fi
    rm -f /service/.$1
  else
    echo "not found: /service/$1"
  fi
}
```

daemondown monitor-ping

图 **5.4.5　daemondown** 的使用示例

- **——daemonstat**

显示 /service 下的守护进程的启动日期及自启动以来经过的时间（代码清单 5.4.10、图 5.4.6）。该函数能够使只显示经过秒数的 daemontools 中附带的 svstat 命令的输出更便于阅读。

代码清单 **5.4.10　daemonstat**

daemonstat() {

```
local -a ds
if [ $# -gt 0 ]; then
  for i in "$@"; do
    ds[${#ds[@]}]=${i#/service/}
  done
else
  ds="*"
fi
cd /service/
svstat ${ds[@]} | \
  while read daemon state dsec pid sec dumy; do
    [ "$state" == "down" ] && sec=$dsec
    printf "%-20s %4s %8ds = %3ddays %02d:%02d:%02d, since %s\n" \
           $daemon $state $sec                                   \
           $((sec / (60* 60* 24) ))                             \
```

```
        $(( (sec / (60* 60 )) % 24 ))                          \
        $(( (sec / 60) % 60 ))                                  \
        $((sec % 60))                                           \
        "$(date -d "$sec seconds ago" "+%y/%m/%d %T")";
done
```

}

```
# daemonstat
dhcpd:          up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
dnscache.in:    up 5227391s = 60days 12:03:11, since 07/10/30 13:46:15
dnscache.lo:    up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
qmail:          up 5637954s = 65days 06:05:54, since 07/10/25 19:43:32
qmqpd:          up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
smtpd:          up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
stone:          up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
tinydns.ex:     up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
tinydns.in:     up 7417649s = 85days 20:27:29, since 07/10/05 05:21:57
```

图 **5.4.6　daemonstat** 的使用示例

5.5　网络引导的应用 **——PXE、initramfs**

5.5.1　网络引导

网络引导（Network Boot）是指在设备启动时从网络上获取必要的资料，以在本机完成启动工作。一般情况下设备启动是由 BIOS 从本地的硬盘或 CD-ROM 等外存设备调取启动时必要的 Boot Loader（引导加载程序）及操作系统内核来完成的。网络引导则与此不同，它是在网络服务器上调取这些必要的启动文件的。

网络引导的特性及优势

若使用网络引导，在机器启动时就无需在本地部署外存设备（例如硬盘等）了。由于网络引导是在网络上取得 Boot Loader 及操作系统内核

的，因此在实际应用中，与通常的情况相比，系统的灵活度会有所增加。特别是在网络引导上应用 **initramfs** [18] 机制时，灵活性更为突出。

> [18] 关于 initramfs 的机制可以参考 Linux 的附带文档。具体可以首先在 kernel.org 等网站获取 Linux 内核发布包，然后通过 linux-2.6.X.X/Documentation/filesystems/ramfs-rootfs-initramfs.txt 文件就可以了解到了。

initramfs 是指在内核挂载到根文件系统及启动 init 之前，仅在内核外部运行初始化的机制。initramfs 的主要功能是在将内核挂载到根文件系统时，将所需的驱动模块加载到内核。

initramfs 的实质是通过`cpio` 工具来收集初始化所需的文件，并对文件进行`gzip` 压缩。在 Boot Loader 将内核读取到到内存时，该文件也会和内核一起被配置在内存上。如果内存上存在 initramfs 的映像，则内核在自身初始化结束之后，挂载根文件之前，将运行 initramfs 中的 /init 程序。在大部分情况下，和通常启动时使用的 init 程序不同，该 init 程序是 shell 脚本。

网络引导中，Boot Loader 在从文件服务器上读取内核时，会一同取得 initramfs。这意味着，即便事先在需要启动的设备上没做任何准备，只要文件服务器准备好了系统的内核及 initramfs，就可以让任何机器启动操作系统。

如果使用网络引导，启动操作系统时就不需要磁盘了，即将操作系统运行所必要的文件系统都存放在本地磁盘之外，这样就实现了典型的无盘系统。在无盘系统中，根系统文件放置在 NFS（Network FileSystem，网络文件系统）[19] 或 MFS（内存文件系统，Memory File System）上。

> [19] 关于在 Linux 操作系统中将 NFS 服务器作为根文件系统使用的详情，在内核的说明文档 linux-2.6.X.X/Documentation/nfsroot.txt 中有详细说明。

通常硬盘都是机器构成要素中故障率最高的部件，所以使用无盘系统的话，设备的故障率就会大幅度减少。

5.5.2　网络引导的行为 ……PXE

以下让我们了解一下网络引导的行为。虽说网络引导可以基于不同的方式实现，但当今 x86 架构中占据主导地位的还是 Intel 开发的 PXE（Pre-

eXecution Environment）[20] 。PXE 实际上就是在网卡（NIC）上植入了扩展的 BIOS 模块。PXE 的引导流程如下（图 5.5.1）：

> [20] ***URL*** http://www.pix.net/software/pxeboot/archive/pxespec.pdf
> PXE BIOS 自身没有相应的配置项目，但使用 PXE 引导需要确保 DHCP 服务器的正确设定。以下网页中有 PXELINUX 相关的说明，请参考。
> ***URL*** http://syslinux.zytor.com/pxe.php

❶ 通常情况下的 **BIOS** 进行初始化操作。在这个过程中浏览扩展的 **BIOS**，登录 **PXE BIOS**

❷ 在启动设备上激活 **PXE**，控制转移到 **PXE BIOS**

❸ 接到控制的 **PXE BIOS** 使用 **DHCP** 获取 **IP** 地址等信息，准备 **IP** 通信

❹ **PXE BIOS** 从文件服务器中获取 **Boot Loader** 进行启动，继续控制。文件服务器的地址和 **Boot Loader** 的文件名可以从步骤 ❸ 中的 **DHCP** 服务器获取

❺ **Boot Loader** 进行启动后，从步骤 ❹ 的文件服务器中获取 **Boot Loader** 自身的配置文件。文件服务器的地址由 **PXE BIOS** 通知给 **Boot Loader**

❻ **Boot Loader** 启动内核后，在此期间若指定了相应的配置文件，就会从文件服务器取得 **initramfs** 文件，并将其和内核一起配置在内存上，其后再将控制移交给内核

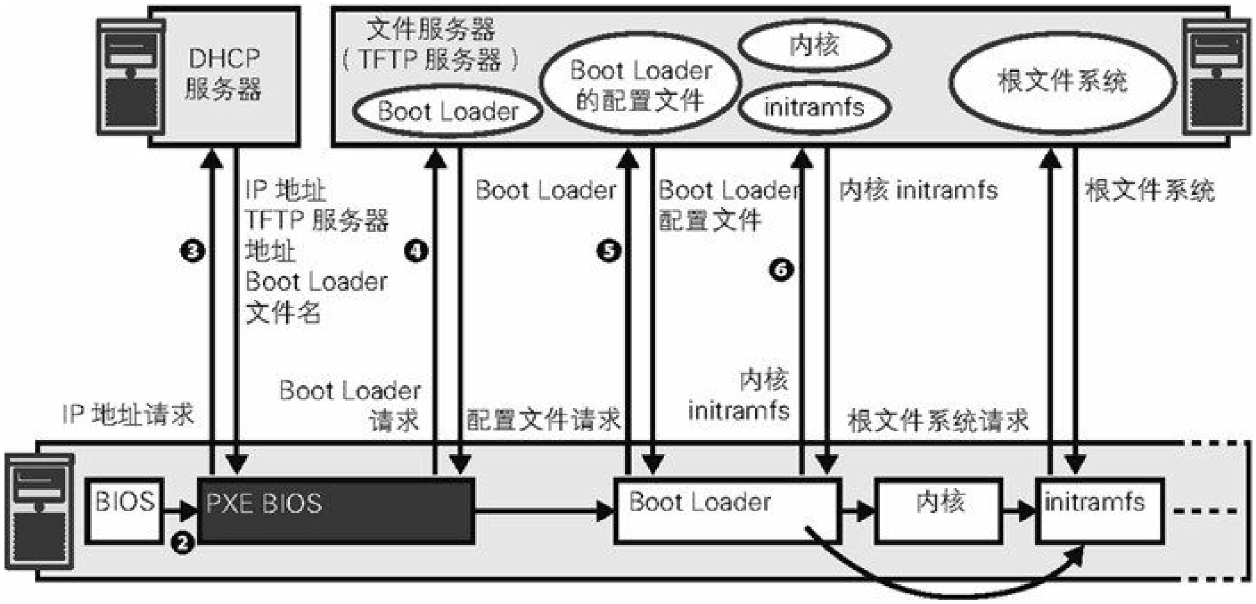


图 **5.5.1**　**PXE** 引导的流程 ※

※ 本案例中使用了 initramfs。

如果移交给内核进行管理，这之后就与平常的启动没什么区别。使用的系统内核也没必要特意支持 PXE，与平常一样使用就可以了。

PXE BIOS 从服务器中获取文件时，使用了 TFTP（Trivial File Transfer Protocol）。这是一个基于 UDP 的简单文件传输协议，无法完成验证的动作。

以下对 PXE 引导的必要事项进行归纳整理：

- 支持 **PXE** 引导的网卡

  近期的服务器设备上安装的网卡应该都支持该技术

- **DHCP** 服务器

  为 PXE 引导提供必要的信息

- **TFTP** 服务器

  PXE BIOS 在取得必要的文件时使用，需要支持 atftpd 等的 TSIZE 选项

- 支持 **PXE** 的 **Boot Loader**

  PXE 引导中需要使用支持该技术的 Boot Loader。有支持 PXE 的 GRUB[21] ——PXEGRUB，以及 SYSLINUX[22] 的 PXE 版 PXELINUX[23]

- 内核

  没有必要为 PXE 引导准备特别的系统内核

- 供根文件系统初始化使用的系统（**initramfs**）

  若要将根文件系统托管到内存文件系统上，则此项必须；除此以外的情况下，使用 initramfs 就能够灵活构建启动系统

- 根文件系统（的内容）

  无论使用什么样的根文件系统，为了确保系统的运行，都需要事先以某种形式准备好所需的文件

> [21] ***URL*** http://www.gnu.org/software/grub/

> [22] ***URL*** http://syslinux.zytor.com/

> [23] ***URL*** http://syslinux.zytor.com/pxe.php

搭建无盘系统的情况下，要在文件服务器上准备供根文件系统使用的文件。如果将 NFS 服务器直接作为根文件系统挂载，则应该先将其设为能够进行 NFS 挂载的形式；若要将内存文件系统作为根文件系统使用，则应该先将其设为便于 initramfs 初始化脚本使用的形式 [24] 。

> [24] 在笔者管理的环境中，根文件系统使用的文件是通过 tar 打包放置到外部不能访问的 Web 服务器上的（根据启动的服务器的目的不同，会准备多个使用 tar 打包的文件）。Initramfs 的初始化脚本通过 HTTP 取得所启动的系统中的 tar 文件，并在内存文件系统上解压。之所以不使用 PXE 使用的 TFTP 而特意使用 HTTP，是因为 TFTP 的文件传输速度比 HTTP 要慢很多。

5.5.3　网络引导的应用实例

下文将以笔者管理的环境为例，向大家介绍一下网络引导的应用。

负载均衡器

负载均衡器（毫无疑问使用了 LVS）是需要关注的重点。每当硬盘遭遇故障时，负载均衡就会停止工作，这会为维护工作带来很多麻烦，所以笔者使用无盘系统来规避此问题。在部署该系统后，至今还未发生过硬件故障造成负载均衡器停止工作的情况。

当然也并不是说不使用硬盘就可以避免故障。万一出现了故障，可以在众多 Web 服务器中挪用一台作为备用机。要使 Web 服务器完成负载均衡器的工作，只需将其作为负载均衡器进行网络引导即可 [25] 。也就是说，仅仅重新启动一台机器就可以完成修复工作。

> [25] 在实际操作中，Web 服务器与负载均衡器可能位于链路层（OSI 的第二层）的两个不同的网络中，鉴于在修复时没空插拔网线，因此在布线时就要注意将所用的设备设计到同一网络中。如果需要分离链路层，可以使用 VLAN 进行分离。在 Web 服务器被作为负载均衡器使用时，为了将设备加入到所需的 VLAN 上，就要变更相应的交换机配置。

数据库服务器 / 文件服务器

数据库服务器 / 文件服务器也可以应用网络引导。但需要注意不能将数据保存到内存文件系统上，因为无论从容量的角度还是从数据的永久性存储的角度来看这都是不现实的，因此需要将数据保存到使用 RAID 冗余的硬盘上，将根文件系统放在内存文件系统上。明明已经存在 RAID 磁盘，还特意把根文件系统放到内存文件系统上，之所以这样做主要是为了简化部署工作。

即便使用了 RAID 阵列的磁盘，还是有可能会发生故障。为了以防万一，需要准备备用机，但由于备用机一般不会用到，如果分别为数据库服务器和文件服务器准备备用机的话，就又有些多余了。

虽然数据库服务器和文件服务器所运行的程序及其行为都有所不同，但在硬件方面两者的结构是相同的，所以备用机可以在两者之间通用。当其中一台设备发生故障时，就将备用机作为故障的系统进行网络引导。这样仅用一台备用机就能为两类设备提供服务，而且无需部署等繁琐操作，能快速完成修复。

维护用的引导映像

网络引导不仅为提供服务准备了必要的系统，也可简化维护工作。

例如，用于服务器设备初始化设置的系统、用于内存测试的系统（例如启动 memtest[26] ）、替换故障磁盘或下架故障服务器时用来删除磁盘上的文件的系统（例如 shred[27] ）等，网络引导提供了基于各种目的的系统。

> [26] ***URL*** http://www.memtest86.com/

> [27] 以普通方式删除磁盘上的数据时，可以通过一些数据恢复手段将数据恢复。shred 工具通过在磁盘上写入特殊设计的 Bit Pattern，使恢复磁盘数据的操作变得更为困难。

5.5.4　构建网络引导

最后介绍配置网络引导服务器时需要注意的几点。

initramfs 的通用化和作用的识别

在使用 initramfs 进行多种系统的网络引导时，若能让 initramfs 自身实现通用化，则该架构更为可取。这是因为 initramfs 在启动内核之后运行，难以调试，维护系统很费工夫。在 initramfs 通用化的情况下也会出现一些问题，例如设备需要进行合适的初始化操作，在此 initramfs 脚本就需要判断将其用于哪个系统。以下有几种策略可供参考。

第一种方法是在内核命令行上将参数传递给 init 脚本。

```
     ↓指定系统
boot: db id =100
          ↑指定数据库服务器
```

内核的命令行用于将参数传递给内核绑定的驱动，若传入多余的参数就会被忽略，并不会产生错误。传递给内核的命令行的字符串可以通过 proc 文件系统在启动后获取。要将启动系统的参数通过内核的命令行传递给 initramfs，可以根据启动的系统数量创建相应个数的 Boot Loader，并在启动时进行选择，或者在每次启动时手动输入。但在任何情况下，Boot Loader 都需要进行对话操作。

还有一个办法，就是 initramfs 根据启动完毕的设备分配的 IP 地址（或者相应的主机名）进行判断。分配 IP 地址是 DHCP 服务器的工作[28]。DHCP 服务器为了对启动的设备分配特定的 IP 地址，会事先调查该设备上的 MAC 地址，并在 DHCP 服务器上配置 IP 地址与 MAC 地址的对应关系。

> [28] 若使用了 IPMI（请参考 5.6 节），则可以通过读取 IPMI 上设定的 IP 地址来进行使用。

或许还有其他方法，但无论采用哪种方法，都要根据自己管理的环境，搭建出使用便捷、方便扩展的架构。

无盘结构中需要注意的事项

搭建无盘结构时，以下几点需要注意。

- ——日志的输出

  第一点是指定日志输出的目的地。一般情况下日志被记录在本地硬盘中，但是无盘结构中本地不存在硬盘，因此若需要在无盘系统中保存必要的日志，可以将这些日志传送到别的设备上进行保存。简便的传送日志的方法有“在文件服务器（NFS）上记录”和“使用 syslog 传送”两种方法。

  网络引导的机器若使用文件服务器，将日志输出到文件服务器上会非常方便。但在这种情况下需要注意两点：第一点是要避免多台设备向同一个文件输出日志；第二点是输出日志的数量。数量太大会阻碍文件服务器的 I/O，有悖于文件服务器原本的使用目的。

  如果不使用文件服务器，使用 syslog 的日志传送功能也十分方便。在这种情况下，不能使用普通的 syslog，而要使用“syslog-ng”[29]，这样发送方就可以在发送日志时使用日志过滤等一些便利的功能。

  那些没有必要保存但是需要在发生故障时查看的日志，可以输出到内存文件系统上。但是在这种情况下，因为内存文件系统的容量比硬盘小，所以要留心所保存的日志的数量。与正常情况相比，存储在内存上的日志应该以更短的时间间隔进行转储（Rotate），并自动删除早先的日志记录，这就是“multilog”的便利之处。

multilog 是 daemontools 附带的程序之一[30]，在接收到标准输入的日志后，multilog 会进行过滤加工，之后再输出。在将日志输出到文件时，multilog 会指定目标文件的最大存放空间，若超出了该空间的容量，就会将较新的日志文件转储，并删除最早的日志。如此一来就保证了日志文件整体的数量不会太大。

- ——文件的变更管理

第二点是根文件系统所使用的文件的变更管理。在将内存文件系统作为根文件系统使用的情况下，即便在启动状态下变更文件，重新启动后也还是会还原。因此，在变更文件的时候，同时要变更文件真正的实体。如果不这样做，在发生故障的情况下为了解决问题而重新启动系统时，就会又遇到别的故障。为了不造成这样的麻烦，应该明确相应的实施步骤。

在笔者管理的环境中，若正在运行的系统中有需要更新的文件，笔者会在更新后确认其行为，并将其复制到 Master 上。为了以防万一，对旧的 Master 也会进行相应的备份操作。另外，笔者还准备了专门的脚本，以使这样一连串的作业实施起来更加简单。

> [29] 有关 syslog-ng 的详情请参考 5.7 节。

> [30] 有关 daemontools、multilog 的详情请参考 5.4 节。

Master 文件的安全性

Master 文件的安全性是很有必要留意的。根文件系统被托管在内存文件系统的情况下，需要准备根文件系统的 Master 文件。该 Master 中不能包含密码或 SSH 密钥等一般用户无法访问的文件。这是因为 Master 文件在机器启动时，会从文件服务器中进行复制。此复制操作由 initramfs 进行，initramfs 能够进行复制也就意味着一般用户也可以进行复制。

而且对从文件服务器的复制进行认证是没有意义的。这是因为为了让验证通过，initramfs 中必须包含相关的认证信息，而通过 TFTP 就可以获取 initramfs 包，因为 TFTP 中是不需要进行认证的。

5.6　远程维护 ——维护线路、**Serial Console**、

IPMI

5.6.1　轻松实现远程登录

追求高适用性的服务器在大多数情况下都会被配置在数据中心。但是系统管理者不会常驻在数据中心，特别是小规模的网站更是如此。服务器的配置地点和平时系统管理者所在的地点相分离，每当需要维护服务器时系统管理者都要跑到服务器的配置地点，这样无论从时间还是从金钱上来说都是一种浪费。因此，通常使用 SSH 等远程维护工具来完成一般的管理操作。

但是在发生故障的情况下，并不一定能进行远程登录。因为远程登录是以系统正常启动为前提的。本节中将介绍在出现故障或系统无法运行的情况下实现远程维护 （Remote Maintenance）的方法。

5.6.2　网络故障的应对

首先介绍一下网络故障时进行远程维护的办法。远程登录位于数据中心的服务器设备时需要使用相应的网络线路，该网络线路就是供服务使用的商用线路。这条商用线路连接着路由器，路由器通过网络交换机（Switching Hub）和服务器进行通信，因此对网络故障的应对还要分为对商用线路和路由器故障的应对，以及对网络交换机故障的应对。

维护线路

为了在商用线路和路由器（三层交换机）发生故障的时候也可以进行远程维护，需要准备一个其他系统的线路。这里把该线路称为“维护线路”。大家可能以为另行准备维护线路的成本会很高，而事实上却很便宜。因为只有在商用线路不能登录时才会启用维护线路，一般情况下使用家用低档次的线路也可以 [31] 。而且不仅在出现故障的情况下，在平常传送大量文件等使用商用线路会对服务造成恶劣影响的情况下，也可以充分利用维护线路，因此维护线路绝不会被浪费。

> [31] 商用线路与维护线路如果使用了不同系统的线路，就不可能出现同时无法使用的情况。

在笔者管理的环境中，维护线路的结构如图 5.6.1 所示。线路是 NTT 的 BFLET'S（+ 该套餐的公网 IP 是固定 IP）。BFLET'S 的光网线路是从

ONU（Optical Network Unit，光网网络单元）的终端上引出的 LAN 网线，该 LAN 网线并非直接连接到路由器，而是通过集线器（Hub）连接到数台服务器上。该集线器与商用线路使用的集线器在物理上完全不同，商用线路的集线器禁止使用 VLAN 进行分割，这是因为如果集线器发生故障，整个结构都不能使用了。

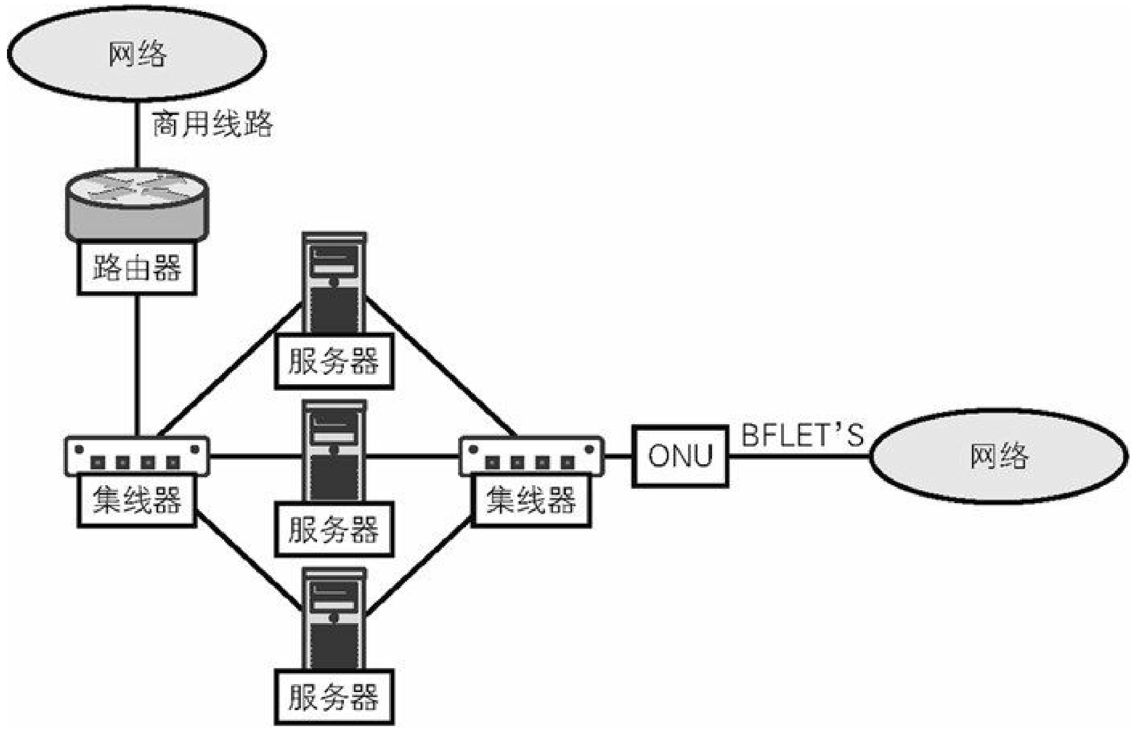


图 **5.6.1**　维护线路的结构案例 ※

※ 左侧是商用线路，右侧是维护线路。

经由集线器连接到 ONU 维护线路的服务器之中，实际上有一台是接入了 Internet 的。正因为该服务器是直接连接到外网的，因此可以不受商用线路或内网故障的影响，在任何情况下都能登录。另外，之所以采用集线器将多台服务器连接到 ONU 维护线路，主要是考虑到在变更接入 Internet 的服务器时，通过该架构可以不用特意赶赴数据中心去插拔 LAN 网线，只需通过控制维护线路，就能解决问题了。

交换机故障的应对

即使在使用商用线路不能远程登录的情况下，通过维护线路也可以登录一台机器。从这台机器登录到另一台机器时需要使用到交换机。接下来

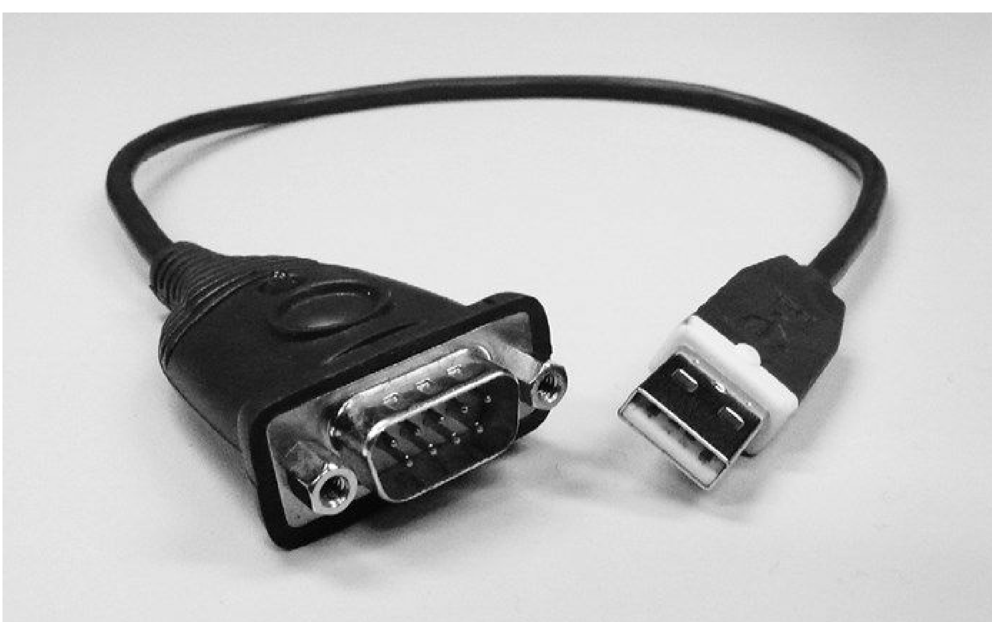
就介绍一下当交换机发生故障时的应对方法。

交换机发生故障的原因主要有两个：第一个是交换机自身的原因，这种情况下应对的方法就是重新启动交换机；第二个是传送给交换机的报文的原因，这种情况的应对方法是切断发送方连接的端口。

网络交换机有智能交换机和非智能交换机两种类型。在使用非智能交换机的情况下，不能远程进行这些操作；在使用智能交换机的情况下，可以登录交换机自身，从交换机的配置接口重启交换机或切断端口。

登录交换机的方法一般有两种：借助 Telnet、SSH 在网络上登录，或通过 Serial Console（串口控制台）登录。在进行一般的操作时会使用前者，即网络登录，因为它的响应很好。但在发生故障导致不能使用网络登录的情况下，则可以通过 Serial Console 登录。关于 Serial Console 的具体内容我们将在下节讲解，这里仅对 Serial Console 的连接方式进行说明。

为了在商用线路发生故障时也能够登录，就需要确保设备连接到维护线路。因此这里将交换机的 Serial Console 也设置为可以从这个设备访问。为此就需要把所有交换机的串行接口（Serial Interface）都连接到该设备上，但一般情况下设备上最多有两个串行接口，因此如果要连接到三台以上的交换机，就需要使用 USB- 串口转换连接器（照片 5.6.1）等设备。



照片 **5.6.1　USB -** 串口转换连接器

5.6.3　Serial Console

至此我们就能应对网络故障了。接下来介绍一下在设备故障导致不能通过网络登录时，以及设备重启时进行远程维护的方法——**Serial Console** [32] 。

> [32] Serial Console 在 Linux 中的使用可以参考“Remote Serial Console HOWTO”文档的说明，该文档暂时没有中文版。
> ***URL*** http://tldp.org/HOWTO/Remote-Serial-Console-HOWTO/

Console（控制台）具体来讲就是键盘和显示器，即输入与输出。在类 UNIX 操作系统中，大部分的管理操作都无需使用 GUI，只需通过文本的输入输出就能完成控制操作。而在 Serial Console 中，则没有使用显示器和键盘，而是通过串行接口将其他设备连接到想要管理的目标设备，进行文本的输入及输出的 [33] 。

> [33] 在工作站普及之前，一台 UNIX 设备连接多个 Serial Console 装置（Dumb terminal，非智能终端）的情况很常见。现今虽然非智能终端已经没有了，但是其功能通过软件得到了实现和应用。

Serial Console 中一般使用 RS-232C 的接口。RS-232C 的通信是一对一的。与以太网不同的是，这里使用一根网线只能完成与一台设备的通信，因此通信双方的两台设备就成为了一对。当需要通过 Serial Console 登录某台设备时，首先需要远程登录与之成对的设备，之后再在目标设备上登录。虽然使用起来有些麻烦，但由于目前大部分服务器中都有一两个 RS-232C 的端口，因此只要准备好网线 [34] ，之后再对软件进行配置就可以使用了。

> [34] 一般使用名为“Serial Cross Cable”的网线。另外，RS-232C 使用的端口分为 9 个针脚和 25 个针脚两种，在服务器设备上使用的是 9 个针脚的 RS-232C。

Serial Console 的实现

Serial Console 与 SSH 的不同点就是不将被登录的一端称为服务器。但这里为了便于理解，我们将被操作的一端称为服务器，将进行操作的一端称为客户端。

以 Serial Console 闻名的客户端软件有 cu、ermit、minicom。鉴于 cu 是历史悠久的程序，可能比较难以上手。若单纯使用 Serial Console，minicom 则更好理解。另一方面，cu 或 kermit 不仅能够应用于 Serial Console，也可以通过串口来传送文件 [35] 。

> [35] 使用了名为 xmodem、ymodem、zmodem 的协议。虽说理论上可以向任何目标传送文件，但需要在服务器上准备发送和接收用的相关软件。

随着设备的启动，Serial Console 的服务器 [36] 所承担的功能也会发生变化。以下按顺序分别说明 BIOS、Root Loader、操作系统、gentty 的行为。

> [36] 这里没有具体列举出来，其实上述交换机就是 Serial Console 的服务器。

- **BIOS**

  如果是服务器设备上搭载的 BIOS，则具有“控制台重定向”（Console Redirection）的功能，即在设备启动时，将 BIOS 输出的消息以及 BIOS 的配置画面，输出到指定的串行接口中

- **Root Loader**

  lilo、SYSLINUX、GRUB 等通常的 Root Loader 都支持 Serial Console。只需进行相应的配置（图 5.6.2 ❶），就可以与一般的终端控制台相同，通过 Serial Console 访问 Boot Loader 的控制画面

- 操作系统

  大部分的类 UNIX 操作系统都会在系统启动时执行的初始化脚本输出相应的信息，这里会将此类信息输出到 Serial Console 上。如果是 Linux 操作系统，则可以在内核的参数中指定 Serial Console 作为默认的控制台（图 5.6.2 ❷）

- **getty**

  在类 UNIX 操作系统中，控制台的登录是使用名为 getty[37] 的程序来进行操作的，Serial Console 的登录也同样使用 getty

> [37] 准确地说，getty 通过监控指定的接口，若发现有信息输入，就进行 Login 程序的登录操

作。

```
default=0
timeout=10
serial   --unit=1 --speed=19200 -word=8 --parity=no --stop=1   ←❶
terminal --timeout=30 console serial   ←❶

title Linux (Console Mode)
  root (hd0,0)
  kernel /vmlinuz ro root=/dev/sda3 console=ttyS1,19200n8 console=tty0 ←❷
title Linux (Serial Mode)
  root (hd0,0)
  kernel /vmlinuz ro root=/dev/sda3 console=tty0 console=ttyS1,19200n8 ←❷
```

图 **5.6.2**　基于 **GRUB** 的 **Boot Loader** 与内核配置案例 ※

※ 本例中 GRUB 的版本是 0.99。
❶ 是 GRUB 自身的 Serial Console 配置，❷ 是内核参数。

getty 有很多种类，其中大部分都能够处理从 Serial Console 的登录，但这里建议使用 **mgetty** 。通常一启动 getty，就会锁定监控的网络接口。但在使用 mgetty 的情况下，只需在启动时附加 `-r` 选项，或者在配置文件（mgetty.config）中选定“direct yes”，就可以在不锁定网络接口的情况下正常执行 [38] （代码清单 5.6.1）。据此，仅仅通过一个 Serial 连接，即使是在 mgetty 监控的情况下，也可以使用客户端程序访问成对设备的 Serial Console。

[38] 若启动 login，就会锁定网络接口。

代码清单 **5.6.1**　**mgetty.config** 的例子 ※

```
port ttyS0
  speed          19200
  direct         yes
  blocking       no
  data-only      yes
  need-dsr       yes
```

```
toggle-dtr     n
ignore-carrier no
login-time     10
term           vt102
```

※ 该例中 mgetty 的版本号是 1.1.31-Jul24。

5.6.4 IPMI

如此一来，在服务器不能进行网络登录以及重启操作系统时也可以进行远程维护了。但如果内核失去控制，则从 Serial Console 也不能登录了。在这种时候，如果设备在身边的话，就可以手动按下电源按钮以强制重启设备，但远程的设备则不能这样操作 [39] 。

> [39] 大部分的数据中心都提供了代替进行这一简单操作的服务。但是从请求代为执行到实际执行重启操作中间需要一段时间，而且也不好意思每次都请求代为执行。

在这种情况下，可以通过网络控制周遭设备的电源，这就是 IPMI（Intelligent Platform Management Interface）[40] 。IPMI 是 intel 公司开发的，是一个从软件出发来控制及确认设备电源状态的标准。IPMI 毫无疑问可以在局域网的设备上使用，同时也可以访问网络上的其他设备以实现相应功能。IPMI 是通过在设备上安装硬件而发挥作用的，因此其运行独立于操作系统。进一步说就是 IPMI 不依赖于机器电源的开关状态，只要给设备提供电流，使用 IPMI 就能够从外部控制设备，IPMI 就好像是独立于需要控制的目标设备的小设备。

> [40] 有关 IPMI 的详情，请查阅各硬件的说明文档以及各软件的网站：
> GNU FreeIPMI ***URL*** http://www.gnu.org/software/freeipmi/
> IPMItool ***URL*** http://sourceforge.net/projects/ipmitool/
> ipmiutil ***URL*** http://sourceforge.net/projects/ipmiutil/

IPMI 的功能

笔者在表 5.6.1 中对 IPMI 的功能进行了归纳总结。目前使用的 IPMI 的版本有 1.5 和 2.0，其中任何一个版本都可以控制电源和获取服务器的传感器以及事件日志等信息。IPMI 2.0 引人注目的地方是引入了 Serial over LAN（SoL），这是通过 IPMI 访问服务器的 Serial Console 的功

能。使用 Sol 就可以不受串联线路的限制，从网络上的任意设备都能够访问 Serial Consol。

表 **5.6.1　IPMI** 的功能

1.5	**2.0**	功能
○	○	开关电源、重启
○	○	获取风扇的转速、温度、电源电压
○	○	获取事件日志
○	○	看门狗定时器（WDT，Watch Dog Timer）
△	○	Serial over LAN。在1.5版本中需要根据情况自行安装
×	○	支持VLAN
×	○	通信加密

使用 **IPMI**

要使用 IPMI，首先需要在设备上安装 IPMI 的功能。安装的方法根据设备的不同存在差异。有的设备原本就配备 IPMI 的功能，这时需要先确认该设备使用 IPMI 功能的相关选项（例如所需的端口等），具体可以询问硬件制造商。

用来访问 IPMI 的软件，有的是由硬件制造商发布的，有的是开源的（OSS）。硬件制造商发布的软件有时只能在同一制造商的硬件上使用，这一点需要注意。而开源的软件则不依赖于操作系统及硬件制造商，因此使用起来较为便利。开源的 IPMI 客户端软件有 FreeIPMI、IPMItool、IPMIutil 等。

5.6.5 总结

这里介绍的内容并不全面。但是鉴于引入该技术的成本较低，而且在平常的管理操作中也能够方便地使用，因此引入该技术绝不会造成亏损。当然，也有很多类似的其他技术，例如 Magic SysRq、看门狗定时器、kdump 等，这里就不再详细叙述了。请尝试使用多种技术，以创造出更加便于管理的环境。

5.7 Web 服务器的日志处理 ——syslog、syslog-ng、cron、rotatelogs

5.7.1 Web 服务器日志的分拣·收集

在真正开始整理分流环境提供服务后，为了统计访问或分析故障，使用日志的情况会相应地增加。因为在分流环境中是由多个 Web 服务器支撑一个服务的，因此日志也会根据服务器的台数被分流输出。但是从日志分析、保存的角度来看，将这些日志集中在一处是比较理想的。本节就将对日志的分拣、收集进行说明，讲述 Web 服务器（Apache）的日志的分拣、收集方法，该方法对其他日志也同样适用。

5.7.2 分拣与收集

日志的分拣 与收集 是两个不同的概念。这里所讲的分拣是指经常把 Web 服务器输出的日志剔除无关的部分后，将相关内容梳理并归纳整理到一个文件中；而收集则是指定期对各服务器输出的日志进行收集并保存。二者的区别在于各自的目的和行为存在差异。

经常对日志进行分拣的目的在于把握服务器在任意时间点的运行状况。例如当故障发生的时候，就可以对具体哪个机器出现了问题进行确认，还可以粗略地统计服务器的访问状况，即瞬时 PV，以及用户数等关联信息。也就是说，发生了什么、在何处发生的，都可以通过日志进行了解。

另一方面，收集日志的主要目的是统计、分析和保存。服务器运维中最基本的就是对 Web 服务器、AP 服务器的日志进行整理分析，因此需要以日、周、月等多种时间跨度为单位的日志信息。而如果需要的日志分

散在各处，统计起来是非常麻烦的。而且考虑到存储的问题，将日志集中在一处也比较便于操作。

即便都是将日志集中在一处，日志的分拣和收集还是存在区别，这是因为二者的日志的精度存在差异。在 Web 服务器中，随着访问量的增多，单位时间内输出的日志数量也在增加。在访问临时出现高峰时，为了确保分拣到所有的日志并加以保存，保存日志的硬件就需要具备稳定的性能。然而，以应对突发状况为前提准备高性能的硬件，这从成本上来看很不合算，因此这里的分拣日志就不要求精度，并另外使用收集的方法收集在各服务器本地上输出的日志。

分拣、收集日志有很多种方法。例如，不使用文件方式记录而是在数据库中进行记录。若在数据库中记录日志，虽然搜索起来比较方便，但管理的经费也会随之增加，通常我认为这方法吃力不讨好。下面笔者就把在实际管理的环境中采取的办法介绍给大家。

5.7.3 日志的分拣 ……syslog 和 syslog-ng

在对操作日志进行分拣时使用 **syslog** 是很方便的。syslog 的作用是在类 unix 操作系统中建立一个日志分拣中心。接下来就向大家介绍一下使用 syslog 来分拣 Apache 的日志的方法 [41] 。

> [41] 虽然本书中没有介绍，但日志的分拣也可以不使用 syslog，而使用 Apache 的 mod_log_spread 模块来实现，具体在《构建可扩展的 Web 站点》（Cal Henderson 著，徐宁译，电子工业出版社 2008 年出版）中的第 10 章有详细说明。
> 若需了解 mod_log_spread，请参考以下链接：
> ***URL*** http://www.backhand.org/mod_log_spread/

使用 **syslog** 进行日志的分拣

Apache 中除了可以将日志输出的目的地记录在指定的文件以外，也可以将日志转送到已经启动的其他程序上。接收到日志的程序可以根据该程序本身的目的处理该日志文件 [42] 。另一方面，syslog 中包含 **logger** [43] 程序，该程序能够将从标准输入接收的操作日志传递给 syslog。通过配合使用这二者，就可以在 syslog 中输出 Apache 的操作日志。

> [42] Apache 中将日志文件转送到其他程序的方法请参考 Apache 文档的 CustomlogDirective 项的说明。

43 有关从 Apache 接收日志文件的 logger 的详情，请参考附带的 man 帮助指令。

在 syslog 中分拣的日志都被设定了程序模块 （Facility）和优先级（Priority）[44] 。syslog 能够据此识别日志，并将需要的日志输出到指定文件上，或将日志转送到其他设备的 sysloy 上。即使是使用 syslog 将 Apache 中输出的操作日志分拣到特定的一台机器上，也要使用程序模块和优先权。也就是说，首先使 syslog 能够识别 Apache 中输出的操作日志，然后再把需要的日志转送到日志服务器（图 5.7.1）。

44 syslog 中的程序模块，换句话说就是范畴（Category）。事先设定好几种程序模块及优先级，通过 syslog 输出日志的程序会在输出时从中选择合适的一种。程序模块的例子有 kern（用于内核）、mail（邮件相关）、daemon（用于各种守护进程）等。优先级的例子有 debug、error、emerg 等。

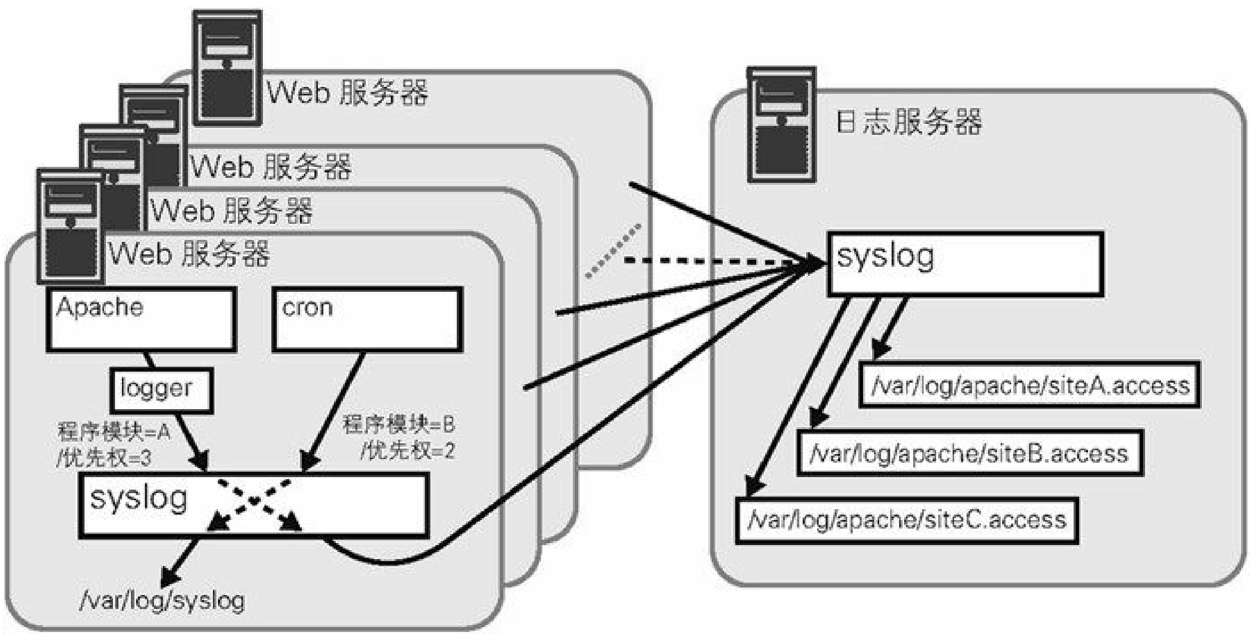


图 **5.7.1**　使用 **syslog** 分拣日志 ※

※ 在需要分拣多个站点的日志的情况下，各个站点的日志的输出目的地不同。

syslog 有时可能会无法正确获取操作日志。当同一个日志被连续输出时，syslog 还是会坚持将其分拣归纳到一个日志中，这种方式不适用于有严密性要求的情况，而且在输出大量的日志时会造成磁盘 I/O 的负担。因此使用 syslog 分拣的日志只在发生问题时查找出现问题的设备时使用，或者被单纯被用于观察站点的趋势。

syslog-ng

假设现在运营了多个站点，要分别对每个站点的日志进行分拣。在类似这样的情况下，分别为各个站点分配程序模块和优先级会花费很多精力。此外，因为程序模块和优先级的组合是有限的，所以能够配置的数量就会有一个上限。可以的话应先解决 Web 服务器使用的程序模块和优先级问题，但是这种情况下各个站点的日志混在一起是非常麻烦的。总之，理想的方式是使用一个程序模块和优先级来传送日志，在输出日志的时候使用别的信息进行区别，最终实现将不同站点的日志输出到不同的文件中。

这些可以通过 **syslog-ng** [45] 来实现。syslog-ng 是 syslog 的升级版。与 syslog 相比，syslog-ng 增加了很多便利的功能，如日志过滤、日志转储、自动生成输出日志的目录等。这些便利的功能之一就是"宏"。宏可以展现与日志有关的元信息，这样就能从宏观角度去了解这些日志所包含的具体信息。syslog-ng 中提供的宏能够表示当前时间和程序模块与优先级、输出日志的主机、输出的日志中设定的标签（程序名）等。在 syslog-ng 中通过使用宏，能够设置输出日志的文件名。也就是说，如果使用表示标签的宏作为输出目的地的文件名，就可以使带有同一个标签的日志在同一个文件中输出。日志的转储也可以通过在文件名中使用表示日期的宏来实现（图 5.7.2 ）。

> [45] syslog-ng 是在以下链接中发布的。
> ***URL*** http://www.balabit.com/network-security/syslog-ng/
> 与正统的 syslog 相比，syslog-ng 的配置文件的编写语法可能比较难以掌握。以下是有关 syslog-ng 的中文介绍及应用实例。
> ***URL*** http://baike.baidu.com/view/3426564.htm
> ***URL*** http://www.oschina.net/question/54100_32993

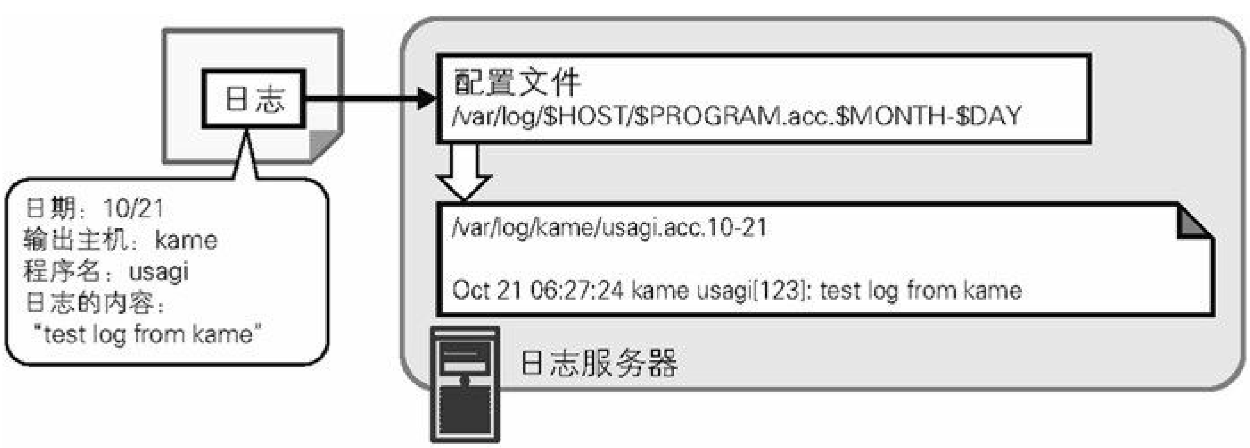


图 **5.7.2　syslog-ng** 的使用示例 ※

※ syslog-ng 可以在配置文件中使用宏指定文件名。

5.7.4　日志的收集

收集操作日志的最重要的目的是保存并分析操作日志。大部分情况下，日志分析是一天进行一回。因此收集操作日志也是一天一次，在早晨服务器的负荷比较低的时候，对每个机器前一天的操作日志进行收集。在笔者管理的环境中，收集的同时也会将各机器上的以前的日志删除。而且还会将日志服务器上的以前的日志进行压缩 [46] 。

> [46] 这样一连串的动作都可以通过脚本实现。在编写脚本时，需要注意发生错误时的记录及修复工作。这是因为如果日志解析程序检测不到日志收集失败，就会将只解析了一半的结果进行报告，因此就会造成混乱。

除了每天进行日志分析之外，日志分析也可以按照一定的时间周期进行，如每周、每月进行一次。而且还可以从新的角度去分析过去的日志。因此这就需要随时都可以马上看到以前的日志。在这里需要留意的是，未压缩的日志会占用大量的磁盘空间，所以一定要对以前的日志进行压缩。即使这个压缩处理需要花费较长时间也没有问题，重点是要尽可能地将其压缩得小一些 [47] 。笔者的管理环境中是使用`bzip2` 来压缩并保存 Web 服务器、AP 服务器的日志的，使用约莫 500GB 的磁盘空间就可以很好地对几年的日志进行保存。

> [47] 在日志的文本数据中，由于已经确定好了输出的字符串模式，因此压缩率比较高。例如某网站的 Apache 的日志，使用 bzip2 的最小压缩选项，就可以将其压缩至源文件的 1/10。

Apache 日志的转储 **……cron** 与 **rotatelogs**

鉴于每天都在收集日志，若将 Web 服务器输出的日志按天进行划分，查看起来应该会非常便利。因为 Apache 本身并不拥有日志转储功能，所以为了将 Apache 输出的日志按天进行转储，就要依赖于外部的程序。这里有两种方法。

第一种方法是使用 **cron** 对日志文件重命名及重新启动 Apache。因为在 Apache 工作的时候，日志文件处于打开的状态，仅仅在中途重命名日志文件是不能实现转储的，所以要在重命名之后立刻重新启动

Apache，才可以实现日志的转储。

第二种方法是使用 Apache 附带的 **rotatelogs** 程序[48]，即在日志的分拣一节中提到的，与向其他程序转发日志的功能配合使用的程序。rotatelogs 将接收的日志写入文件，并且具备在写入日志时将日志文件转储的功能。

> [48] rotatelogs 是实现 Apache 日志转储的方式，使用方式见 Apache 的 man 帮助，也可参考下面的页面：
> ***URL*** http://httpd.apache.org/docs/2.0/programs/rotatelogs.htm

遗憾的是，无论使用哪一种方法，都不能完全实现日志的实时转储。使用 cron 的情况下，重新启动 Apache 时肯定会发生时间上的波动。虽说使用 rotatelogs 的方法比起使用 cron 的方法在转储时具备更严密的时间判定。但是因为 Apache 从收到请求到生成日志并传递给 rotatelogs 存在延迟，例如即使事先设定在 0 点进行转储，也有可能发生昨天访问的日志被输出到今天的日志文件上的情况。

5.7.5　日志服务器的作用与构成

日志服务器的作用就是对日志进行分拣和收集，并将收集的日志进行保存。除此以外，在对其他收集来的日志进行分拣和分析时，也会使用到日志服务器。日志的分拣收集、旧日志的压缩、日志分析等都是比较重要的处理，因此提供服务的服务器不能同时承担日志服务器的工作。另外，日志的保存通常都需要容量足够大的硬盘。因此最好能够提供专门的设备作为日志服务器。

笔者的管理环境中准备了主用和备用两台日志服务器。备用的日志服务器中放置了备份的日志文件，还承担了在主用的服务器出现故障时代替它的作用。另外，对于按月或按年分拣的大量的日志文件来说，分析是十分必要的，但分析会耗费相当程度的资源，为了不影响日常的日志分拣，可以将大规模的日志分析放到备用的日志服务器上进行。

无法准备专门的备用服务器的情况下，就需要将日志文件转移保存到别的设备上。该设备不一定要在数据中心，使用维护线路将日志文件传送到办公室的设备上也是一种方法。在这种情况下，需要注意日志文件中包含的敏感信息。

5.7.6 总结

本节笔者以自己管理的环境为例介绍了日志的分拣和收集。根据日志的内容和操作方法，对日志的要求也有所变化。例如在一些站点中可能需要实时获取 PV、用户数等。在大规模的站点中，鉴于日志的数量过多，会出现保存相对麻烦的问题。当然，根据站点的不同，对于日志的要求肯定是千差万别的。即使是同一站点，随着站点自身的成长，对日志的要求也会发生变化。因此，获取日志的处理也要根据具体情况来灵活应对。希望本节的内容能够对您有所帮助。

第 6 章　服务后台 ——自律的基础设施、稳健的系统

6.1　Hatena 网站的内容

6.1.1　Hatena 的基础设施

Hatena[1] 是一个提供博客（Hatena Diary）、书签（Hatena Bookmark）、百科搜索等 Web 服务的网站。截至 2008 年 2 月，月均访问量达到了 970 万次。Hatena 基于其自身的理念提供了独特的 Web 服务，同时也建立了一个特殊的基础设施。

> [1] ***URL*** http://www.hatena.ne.jp

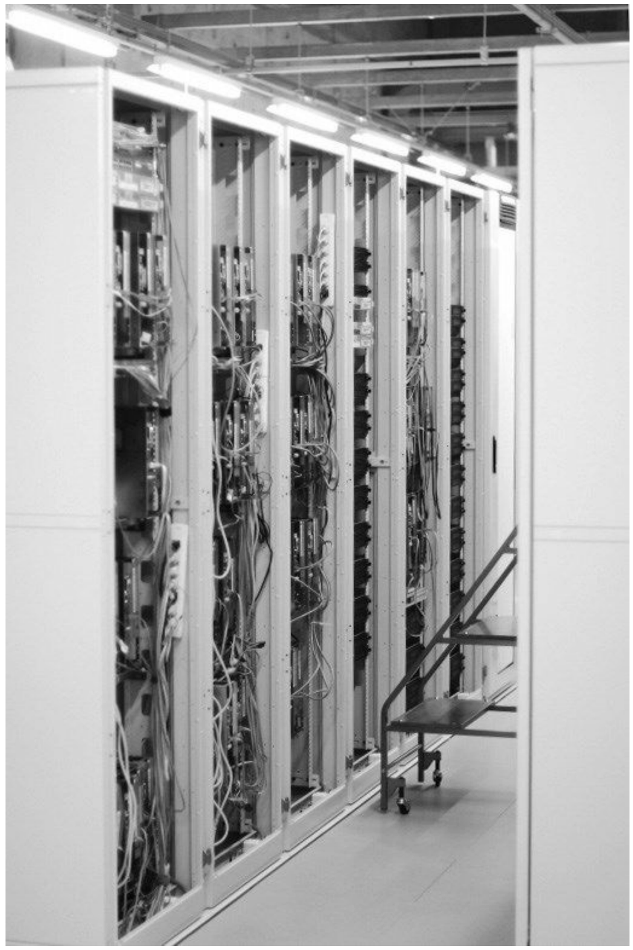
在 Hatena 的基础设施构建方面，始终坚持了“自我原则”和“开源原则”这两个原则（照片 6.1.1 ～ 照片 6.1.3）。“自我原则”是指，不购买现成的服务器，而是自己设计和组装服务器硬件及方案。“开源原则”是指，在服务器上操作的软件几乎全部都是 OSS 开源软件。另外在自行编写软件或为软件附加功能时，也尽可能地利用了开源社区的相关技术。



照片 **6.1.1　Hatena** 网站的服务器



照片 **6.1.2**　配置好的自制服务器



照片 **6.1.3**　数据中心的内部情况

截止到目前，Hatena 的整个基础设施已拥有约 350 台服务器。其中提供服务的基础设施主要由三个层次构成，即反向代理服务器、AP 服务器、数据库服务器，并根据负载和需求添加了高速缓存服务器、文件服务器或批处理服务器。另外还有日志服务器、监控服务器、资源服务器等共享的服务器（图 6.1.1）。

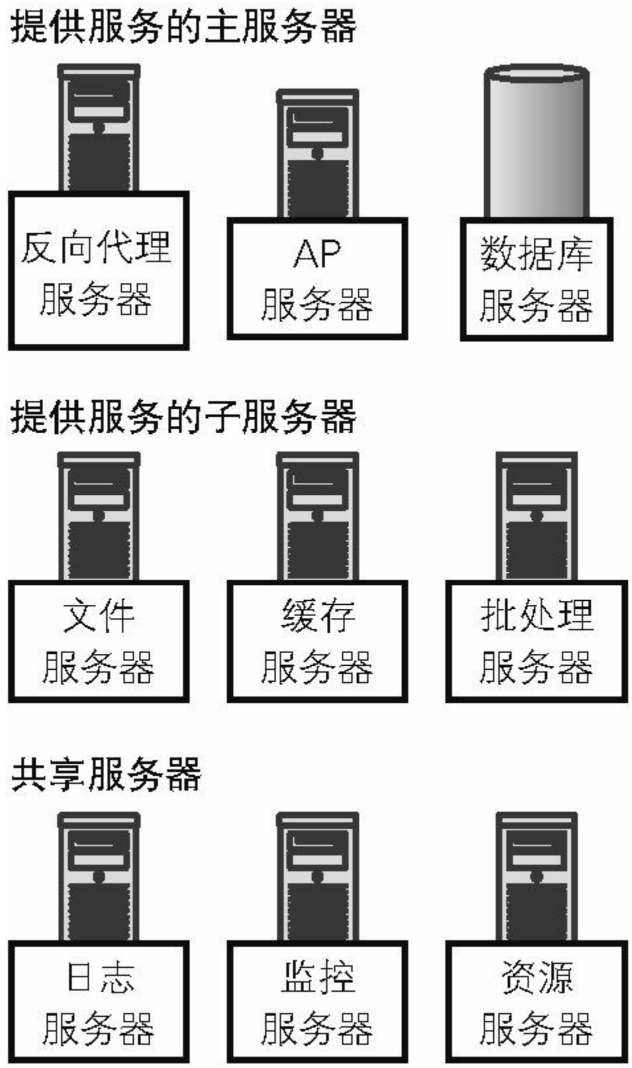


图 **6.1.1**　**Hatena** 网站的服务器构成

正是因为这些服务器集群每时每秒都在工作着，Hatena 网站才能够持续地提供服务。本章将介绍 Hatena 为提高基础设施的稳定性、营运效率、用电效率和可扩展性等所做的努力。

6.1.2　可扩展性和稳定性

对 Hatena 这样每天都要处理海量请求的网站来说，通过进行有效的负载均衡来维持较高的可扩展性是非常重要的。

Hatena 是利用 LVS（IPVS）+keepalived 来实现负载均衡的，在此准备两类服务器，将使用 VRRP 实现冗余的主机集群作为一类（图中的共享

服务器），将各反向代理服务器、AP 服务器、数据库服务器这三个层面的服务器作为另一类（图中提供服务的主服务器）。在 LVS 服务器的存储中不使用普通硬盘，而是使用固态硬盘以努力降低硬件故障率。另外，通过在 LVS 中利用 DSR 路由协议，既可以减少通过 LVS 服务器的流量，而且还能将各层服务器划分为类来把握全局。如果未来访问量进一步增长，也可以根据情况细化各层类的划分以方便管理。

反向代理

反向代理主要使用的是 Apache 2.2。反向代理代理交接了多个后端模块，这里不使用 mod_proxy_balancer，而是使用 mod_proxy 模块来进行反向代理。负载均衡则是利用上述的 LVS 技术来完成的。此外，和 AP 服务器之间通过利用 Squid 实现最大限度的缓存，以减少 AP 服务器和数据库服务器的负载。

对反向代理 Apache 还实施了 Dos 攻击的防范策略，即利用笔者开发的 **mod_dosdetector** [2] 以动态检测 DoS 攻击。引进 mod_dosdetector 之前，每当受到 DoS 攻击时只能手动添加查找攻击源 IP 地址的设定，实际解决起来非常复杂，但通过引入 mod_dosdetector 模块，就可以动态消除攻击，简单的 DoS 攻击基本不会造成什么影响。

> [2] ***URL*** http://sourceforge.net/projects/moddosdetector/

在 Apache 的架构中，针对一个请求，worker 模式下会为其分配线程，prefork 模式下会为其分配进程，而且在等待 AP 服务器响应期间，分配也不会被释放。因此，在收到大量请求的情况下，就算 AP 服务器的负载程度较低，也有可能会使用掉过多反向代理的资源。在此之前，在 Hatena 网站的书签功能中，用户调取书签的 API 就发生过这个问题。该 API 的请求与其他请求相比数量差距悬殊，为了完成 API 的处理，正常页面会出现响应延迟的状况。因此，我们在 Apache 的前端再准备一个不同架构的 Web 服务器 lighttpd 作为反向代理，将 API 和正常的请求分类，使处理效率得以提高。这样优化架构以后就能快速地做出响应了。

数据库

在可扩展性和稳定性方面，数据库容易成为瓶颈。Hatena 全面采用了 MySQL 数据库。初期阶段的 Hatena 使用了 PostgreSQL，之后很快便转到了 MySQL，并沿用至今。2008 年是 MySQL 4.0 系统和 MySQL 5.0

系统共存的状态。笔者个人希望不再继续使用淘汰的 4.0 系统，而是全部迁移到 5.0 系统，但因为 4.0 系统和 5.0 系统时间戳的记录不同，要想全部迁移到 5.0，必须重写应用程序，所以不太容易进行。

MySQL 采取了主从（Master-Slave）同步结构，Slave 集群也通过 LVS 实现了负载均衡，当其中一台 Slave 发生故障时，会自动阻止请求发往遭遇故障的服务器，从而增强了可用性。

另外，已经迁移到 MySQL 5.0 的数据库采用了多主（Master）结构，通过 keepalived 切换 Active/Standby，实现了主数据库的冗余（图 6.1.2）。多主结构的具体设置如代码清单 6.1.1 所示。通过此设置，多个 Master 就变成了“互为主从”的关系，彼此的写入会被传递到对方。但在采用多主结构的情况下，指定 auto increment 的地方可能会出现 INSERT 冲突的问题。因此 MySQL 在进行 autoincrement 时，需要指定增长量（auto_increment_increment）和偏移量（auto_increment_offset），这样就能够在多主结构下正确使用 auto increment 了。

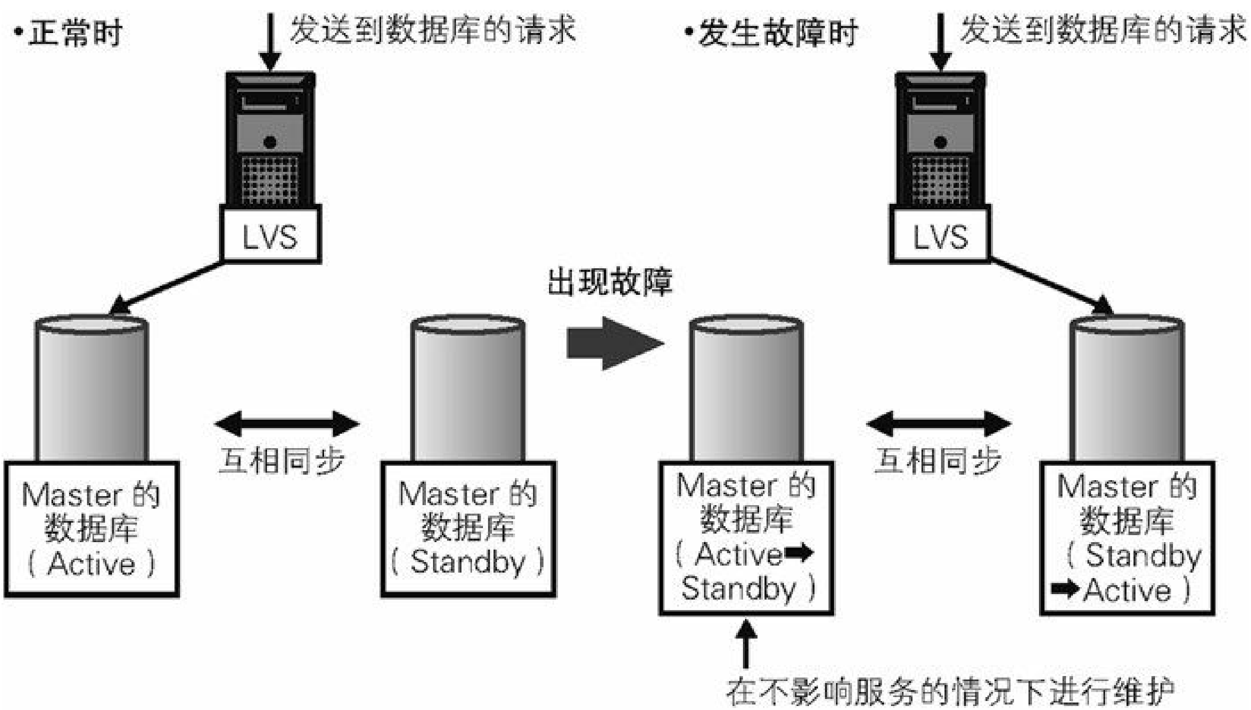


图 **6.1.2**　**MySQL** 的多主结构

代码清单 **6.1.1**　多主结构的配置

```
●服务器 A

 （192.168.1.1）
server-id=1001
master-host  = 192.168.1.1
master-user  = repli
auto_increment_increment = 16
auto_increment_offset    = 1

●服务器 B

（192.168.1.2）
server-id=1002
master-host  = 192.168.1.2
master-user  = repli
auto_increment_increment = 16
auto_increment_offset    = 2
```

此外，即使在不使用 auto increment 的数据库结构中部署了 Active/Active（多主）配置，根据应用程序行为的不同，也有可能会因为 Duplicate Entry 错误而导致同步中断。因此，需要在每个主服务器上安装 keepalived，根据 VRRP 来配置 Active/Standby，让执行修改类的 SQL 语句只在一方进行处理。据此，不管应用程序的行为如何，在任何情况下都可以稳定运行了。

为数据库的表添加列，或者执行表的优化操作时，在传统的主从结构下，进行这些维护必须停止服务，而在迁移到多主结构的数据库中，则可以在不影响服务的正常提供的情况下进行，可以说向前迈出了一大步。

在主机Ⓐ（Active）和主机Ⓑ（Stand-by）的多主结构中，具体程序如下所示：

- 在主机Ⓑ（**Stand-by**）上停止 **keepalived**
- 分别在主机Ⓐ 和主机Ⓑ 上运行 **SLAVE STOP**，停止同步
- 在主机Ⓑ 上执行添加列等维护操作
- 在主机Ⓑ 上运行 **SLAVE START**，恢复从Ⓐ 到Ⓑ 的同步

- 等待主机Ⓑ 的同步赶上主机Ⓐ
- 启动主机Ⓑ 的 **keepalived**，停止主机Ⓐ 的 **keepalived**，将主机Ⓑ 恢复为 **Active**
- 在主机Ⓐ 上执行 **SLAVESTART**，恢复从Ⓑ 到Ⓐ 的同步
- 等待主机Ⓐ 的同步赶上主机Ⓑ
- 启动主机Ⓐ 的 **keepalived**，将主机Ⓐ 恢复为 **Active**

这是一个复杂的过程，但这样就可以在服务不停止的情况下进行若干次维护。不过这里必须保证所实施的维护的内容与并行运行的 Active 的更新内容不冲突。例如，在添加列到现有表时，如果被添加列的内容全部都可以为默认值，就没有冲突问题；另一方面，在进行删除列的维护时，在 Active 一方对列的删除做出反映之前就运行 Insert 语句的情况下，若 Insert 语句中包括该列的项目，就会在“等待主机Ⓑ 的同步赶上主机Ⓐ ”时发生同步错误。因此像后者这样，在恢复同步时因错误导致同步终止的情况下，必须中断服务进行维护。

多主的另一个问题是，当访问量增加时，仅靠两台 Master 将无法处理访问，而必须增加 Slave。各 Slave 都从属于某一台 Master，如果自己从属的 Master 出现故障，而只有一个 Master 运行的话，就会导致更新信息无法传递给出现故障的 Master 下所挂载的 Slave 数据库。因此，当出现故障时只能通过别的方式传递更新信息，或者停止出现故障那边的 Slave 数据库，这个问题目前还没有找到很好的应对措施。

如果能够采用前者的方式，从尚未出现故障的 Master 获得更新信息当然非常理想，但是从 MySQL 的同步机制上来看这比较困难，因此如果要应对的话，也只能采用后者的方式，即停止出现故障那边的 Slave 数据库，但这样做会导致处理能力减半，如果服务器资源不够充足，在高峰时间就很可能发生更为严重的故障。虽然这一问题现在还没有到不得不解决的地步，但在不久的将来还是要必须面对的。

文件服务器

和数据库一样，文件服务器的可扩展性和稳定性也容易遭遇瓶颈。在 Hatena，文件服务器均已使用 DRBD ＋ keepalived 进行过冗余处理，通

过使用 lighttpd 的反向代理优化的 API，还有通过 Squid 优化的缓存，从而在冗余和高速之间取得了平衡。另外在 DRBD 上冗余的块设备使用 OCFS2（Oracle Cluster File System for Linux2）[3] 集群文件系统进行了格式化，通过使用集群文件系统，就可以使 DRBD 的 Active 一方提供服务，Backup 一方进行备份，这样日常的备份工作所造成的 I/O 压力就不会影响到服务的提供了。

> [3] ***URL*** http://oss.oracle.com/projects/ocfs2/

6.1.3　提高运营效率

投入新服务器到基础设施生产环境时，应注意以下几点：

- 硬件的组装和配置
- 安装系统
- 安装并配置应用程序所需的库等
- 配置监控等其他基础设施所需的运行准备
- 根据服务器的作用部署应用和配置数据库
- 部署负载均衡器，以启动生产环境

根据应用的行为进行配置、完成数据库的数据同步操作时，通常都会相当耗时。但这之外的部分，基本都实现了流水线化的自动作业，例如从硬件的安装到实际投入使用，几乎不需要管理员参与。

下面简单地介绍一下其流程。

安装 **Kickstart**

Hatena 的服务器基本上都是从基础零件组装的。这部分纯粹是靠人海战术，确实需要一定的时间。组装完服务器并设定 BIOS 之后，首先用 Kickstart 进行操作系统的安装，除了通常所需的最低限度的操作系统的安装之外，还应该进行 LDAP（Lightweight Directory Access Protocol）的配置，以及使用 autofs 进行用户登录方面的设定，此外还有 Puppet

的初始配置，另外还可以根据需要配置将自己的 IP 地址发送给工程师进行沟通所使用的 IRC 消息频道。鉴于这里会安装大量的软件包，所以需要一定的时间。

这样一来，在 BIOS 的设置完成后重启设备，就完全可以进行远程操作了。

软件包管理和 **Puppet**

通常在进行“安装并配置应用程序所需的库等”以及“配置监控等其他基础设施所需的运行准备”等工作时，需要花费相当多的人力和时间。但这里通过建立或引入以下两种工具：

- 全部使用 **rmp** 安装包或使用 **yum** 一键安装

- **Puppet**（自动化配置管理工具，具体请参考 **5.3** 节）

就几乎实现了自动化，可以大幅度减少配置花费的时间。

前者是指在希望使用 MySQL 的特定版本或 Apache 需要一些非标准库时，使用 rpm 包或通过 yum 工具进行安装。比较麻烦的是大量的 CPAN 模块，这里通过分析 CPAN 的依赖关系可以自制脚本打包成 rpm，这样 CPAN 模块也几乎可以自动转换为 rpm 包。据此，Hatena 应用所依附的多达 200 余个 CPAN 模块也可以轻松地进行部署了 [4] 。

> [4] 打包为 rpm 后，更新依赖于 CPAN 模块的应用也变得更容易了。

在 5.3 节中介绍的 Puppet，通过逐渐积累运营诀窍，使服务器所需的初始配置实现了自动化。Puppet 经常被用于一些很少需要调整的配置中，例如域名服务器的设置和 sshd 的设置，以及后端服务器和数据库服务器的基础设置。至于各应用程序需要根据负载情况详细修改配置的配置文件，则仅部署初始化文件，其余的设置则依赖人工调整而不使用 Puppet。

从 Puppet 的特征上来看，设置的微调也需要多个步骤的冗余操作。通常需要采用试误法（Try and Error）来尝试多次重写配置，例如 Apache 的 httpd.conf 和 MySQL 的 my.cnf 等配置文件，考虑到配置生效的速度和效率，最好由各个管理员提前编写几个不同情况下的配置文件，届时

直接更换合适的配置文件就可以了。

在服务器的设置完成之后，就能够轻松地部署应用程序并确认程序的运行。其后再更新负载均衡器（LVS）的设置，新服务器的投入工作 就结束了。

服务器的管理和监控

鉴于笔者平时经常会对服务器进行添加、设置和微调等操作，因此需要了解某服务是在哪个服务器上运行的，还需要准确了解各个服务器的具体规格。以前我们的管理团队是通过 Hatena 集团的关键字功能（类似 Wiki 那样的实现）来管理列表的，但由于更新过程过于复杂，很容易出现更新纰漏，例如常常在想使用某服务时才发现该服务其实是在其他服务器上运行的。

造成这一问题的最大原因是，若改变一个服务器的作用，就会牵连到下述多项需要更正的项目，所以很难避免人为错误。

- 修改服务器管理关键字
- 更改 **Nagios** 监控的设置
- 在视图监控工具 **MRTG**（**Multi Router Traffic Grapher**）[5] 上修改配置
- 更改应用程序部署的目标列表
- 更改 **LVS** 的配置

[5] ***URL*** http://oss.oetiker.ch/mrtg/

接下来开始自行建立服务器管理工具。我们的目标是：只需修改服务器管理工具上面的数据，相应的配置就会自动得到修改。在目前情况下，通过利用 MRTG 的视图化接口，在自行建立的服务器管理工具上加入视图监控等功能，这样一来就能在部署时避免更新目标对象列表了，即无需手动修改配置文件，通过视图端就能完成监控操作。今后视图端还会添加 LVS 和 Nagios 的控制台。

服务器管理工具能够输出诸如服务器台数以及各种规格的服务器台数统计信息，结合 Hatena Graph 的相关功能，还能够掌握基础设施规模和架构的变革。最近发现 Hatena 的基础设施中依然在使用 Pentium MMX，笔者不由得有些感慨。

该服务器管理工具将作为 OSS 开源软件进行公布。

使用 **Capistrano** 部署

Capistrano [6] 是很有历史的应用程序部署工具了，它几乎能一键完成部署操作。Capistrano 是使用 Ruby 语言开发的应用部署工具，不仅可以应用于 Ruby on Rails 的 Web 应用框架，还能应用于其他比如 PHP 开发的 Web 应用上。通过使用该工具，就可以将需要执行的任何命令发送到多台服务器批量执行，还可以在 shell 中交互式地执行命令，非常方便。

> [6] ***URL*** http://www.capify.org/

在 Hatena，并没有像这样直接使用 Capistrano，而是对 Capistrano 的功能进行了一部分调整。例如通过上述服务器管理工具的 API 获得发送部署等命令的目标服务器列表等。因此，即使基础设施的维护团队增设了服务器，Web 应用程序开发者也可以不考虑这一点，无障碍地进行应用程序的部署和更新等。

6.1.4　用电效率·提高资源的利用率

运行大量的服务器并不断增强其性能，基础设施的运营成本必然会增加。这样就产生了一个问题——是继续使用现有的服务器呢？还是引进效率较高的新服务器呢？

Hatena 不仅重视购买服务器的费用等初期成本的压缩，同时也重视用电成本的压缩。因此，日常会基于以下三个观点进行服务器的调度和配备。

- 在配备和调度服务器时，重视服务器每 **1A** 所发挥的性能
- 尽可能提高每台服务器的性能，根据虚拟化技术分割利用

- 不安装无用的零件

重视服务器每 **1A** 所发挥的性能

在选择服务器时，大多数情况下人们都会比较重视硬件本身的价格，而往往忽视硬件的耗电量。而在服务器制造商提供的参数列表中，往往耗电量写得也并不那么明确。在 Hatena，也有自己组装的硬件，在评定耗电量时，不仅是 CPU 和芯片组，内存、硬盘和电源装置的耗电量也都要一一进行测试，并根据测试结果来选择。

对于现有的、已经在生产环境中的服务器，也会通过更换一部分零件来提高性能，降低耗电量。最近的案例中，最具代表性的是从 Core2 Duo 的 CPU 更换为“Core2 Quad”，据此就实现了性能翻倍且耗电量不变的效果。这时据 Core2 Duo 的引入大约只经过了一年的时间，就迅速转变为了 Core2 Quad，目前 Core2 Duo 只保留了全盛时期的 1/3 左右。通过这个调换工作，就能够在不增加服务器的情况下增强服务器的承载量，同时也能控制数据中心的机架成本和电源成本，从而有助于基础设施成本的压缩。

充分利用每台服务器的性能

如上所述，从 Core2 Duo 转换到 Core2 Quad 后，将原本 Core2 Duo 服务器的软件放到 Core2 Quad 服务器上并进行冗余，接下来就产生了另一个问题，那就是不能充分利用现有服务器的性能。鉴于 Core2 Quad 的性能非常高，仅用 1 台就能够处理某种程度的并发量，如果是用于小型

服务的话，就会浪费其性能优势，更何况为了进行冗余还要准备 2 台，除非是相当规模的服务，否则 Core2 Quad 服务器的性能肯定无法被充分利用。鉴于每台服务器的性能还会逐渐提高，这种趋势会变得更加明显。

在这种情况下，虽然使用廉价的 CPU 也是一个解决方案，但是实体 CPU 的价格及耗电量等都不具有优势，因此就会导致为了支撑小型 务而花费较大成本的情况。在 Hatena，通过引进虚拟化技术 **Xen**， 实现了更有效的服务器资源的利用。

使用 Xen[7]，可以实现在 1 台物理服务器（母机）上提供多台虚拟服务器（子机），但子机分配的资源要在母机内存、硬盘和负载容许的条件

下进行。

> 7 ***URL*** http://www.xen.org/

目前 Hatena 的标准配置是，将首先启动的母机的操作系统（Xen 的术语为 **Dom0** ）用于管理，仅为其分配最低限度的硬盘和内存，另外生成若干个子机的操作系统（Xen 的术语为 DomU）。例如，用两台服务器建立 AP 服务器的 DomU 和多主数据库的 DomU，这里准备两台的原因是为了进行冗余，实现当一台服务器出现硬件故障时，也可以分别保留一个 AP 服务器和数据库的 DomU。之所以分离 AP 服务器和数据库，是为了方便在未来服务成长时将各个 DomU 部署到不同的服务器。

此外，数据库服务器的内存容量和 I/O 性能容易成为瓶颈，AP 服务器的 CPU 性能容易成为瓶颈。因此，通过将数据库服务器剩余的 CPU 资源配入到 AP 服务器的 DomU 上，这样就能够充分利用 CPU 及 I/O 资源了。

Hatena 通过这一方式逐步推进了基础设施资源的有效利用（图 6.1.3），目前物理服务器的台数是 350 台左右，虚拟的子机服务器数量要比物理服务器多 20% 左右。

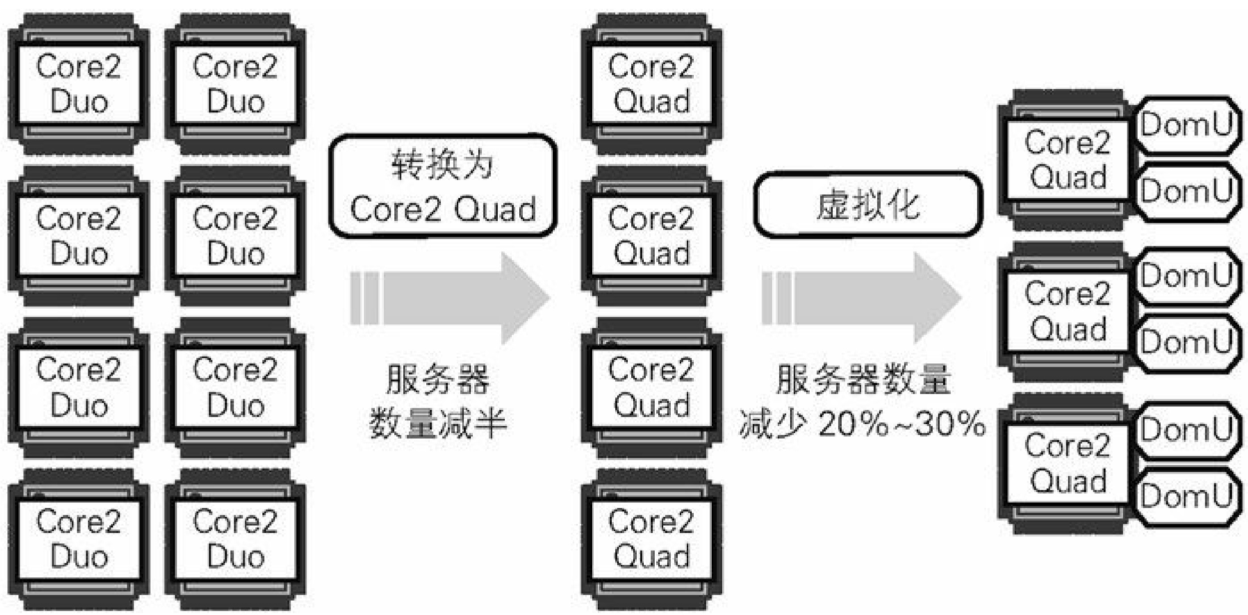


图 **6.1.3**　基础设施资源的有效利用

鉴于每台服务器的耗电量会随着资源利用率的提高而增加，因此单台电

源实际连接的服务器台数要比想象中少。

不安装无用的零件

提高电源效率的另一个方法就是不安装无用的零件。比如，不安装廉价的内存、控制硬盘容量、不组装无用的 RAID 等。现在 Hatena 正在引入原本就没有硬盘的无盘服务器。

无盘服务器，顾名思义，就是没有硬盘的服务器，它的优点是能够减少耗电量、减少硬件故障发生率、容易通过网络引导修改所部署的服务的配置。缺点在于不能自行启动、日志等不能在本地保存、相对于有盘服务器运行不那么稳定等。

Hatena 的无盘服务器有以下几个特点：

- 通过网络引导和 **DHCP** 进行服务器作用的配置
- 基于虚拟化技术的高维护性
- 通过联合文件系统 [8] 和文件服务器进行松耦合

> [8] 即 Aufs。Aufs 是 Another Unionfs 的简称，是一个类似于 Unionfs 的可堆叠联合文件系统。
> ***URL*** http://aufs.sourceforge.net/

实际上，文件服务器启动后，会按照如下流程依次启动：

- 通过 **DHCP**，根据 **MAC** 地址获取 **IP** 地址和磁盘映像的路径
- 在初始 **root** 上获取磁盘映像，并将已经获取磁盘映像的 **root** 作为实体 **root** 进行挂载
- 启动实体 **root**

通常只在启动时才需要和外部文件服务器及 DHCP 服务器等进行连接，在系统启动后，即使切断和这些服务器的连接也可以继续运行。在启动后，若需要部署应用程序等，文件将会被更新，这里所更新的内容会自动写回到磁盘映像中，用户在使用时甚至不会察觉到这是无盘服务器。但是，被分配了相同任务的无盘服务器可能都是通过同一镜像启动的，这一点需要留意。

使用无盘服务器，例如添加新的后端服务器进行负载均衡时，只需改写 DHCP 的配置文件，并启动相应的服务器，服务器就能够自动下载合适的磁盘映像，并将其投入使用。

6.1.5　为了自律的基础设施而努力

在这一两年中，Hatena 的基础设施在负载均衡、冗余、提高资源利用率方面积累了不少经验，现在已经拥有一套能够承受相当规模的访问量的服务器并实现高效运作了。不过还有一些地方需要依靠专业人员的设置，或者需要反复手动进行微调。平时随着并发量的增加，服务数量及类型也在持续增加，在基础设施技术方面，例如在虚拟化技术上，需要调整的地方还有很多。随着运作效率的提升，服务器资源余量会越来越少。因此也很容易想象到在不久的将来，管理者将在这些细节的调整上花费越来越多的时间。

希望今后可以根据这些在已有设备的调试方面所积蓄的经验，使各服务器的调整、增设、撤除逐步实现自动化。例如，监控 Apache 的进程，如果负载增大就增加 MaxProcess，如果进程的内存消耗增加就缩小 MaxProcess 和 RequestsPerChild，这些想必很快就能实现。此外，如果能将无盘服务器的运用很好地推入正轨，似乎也能很容易实现动态增加或减少某一作用的服务器。

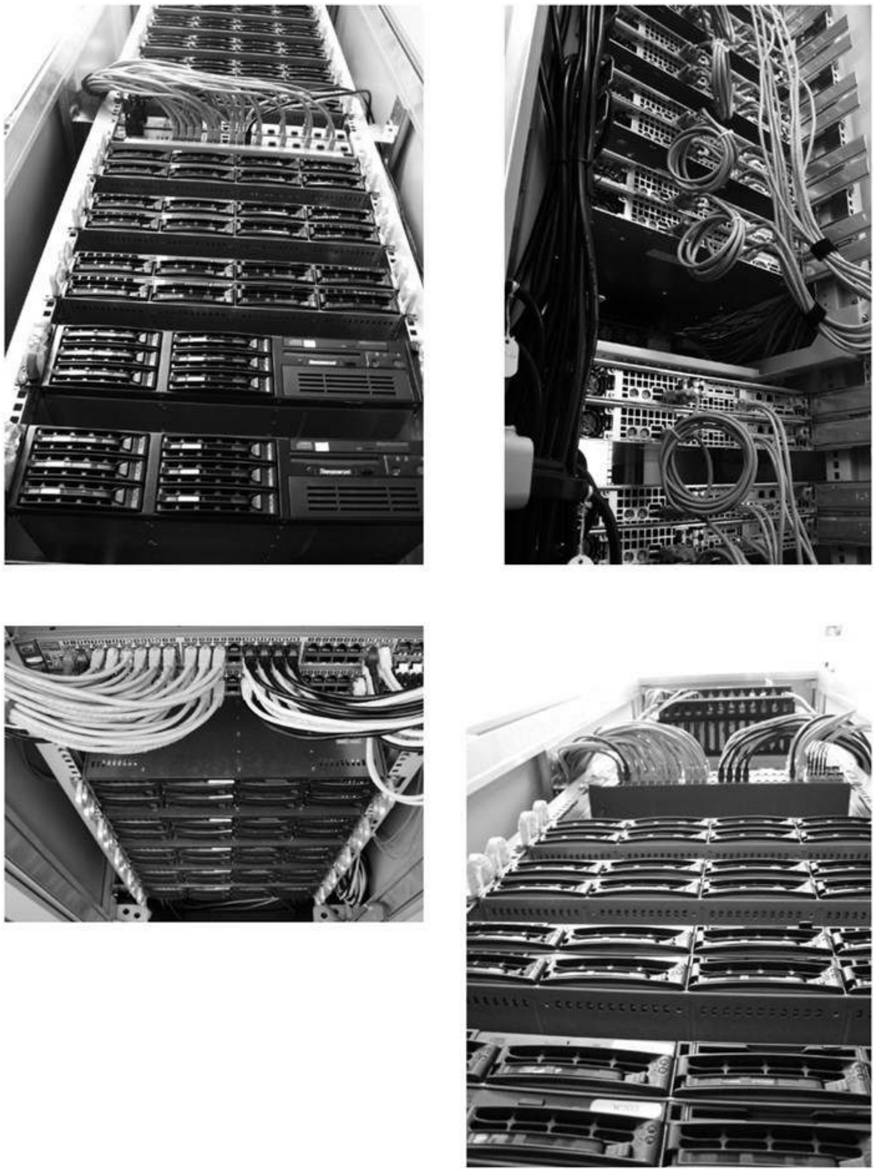
像这样不断地突破手动的自主控制，基础设施就会变得像生物一样。各服务器独立运行，出现故障就自行修复，若负载增加就强化该部分，若长期不使用就让其弱化。当然硬件层面的工作仍需人工进行，但针对可以用软件完成的逻辑部分，则构筑像生物那样自律顽强的“生态系统”。笔者认为，这将是基础设施的终极形态。

6.2　DSAS 的内容

6.2.1　什么是 DSAS

DSAS（Dynamic Server Assign System）是 KLab 公司 [9] 使用的服务器·网络基础设施的统称。目前在东京和福冈的数据中心有 300 多台服务器运行着（照片 6.2.1）。

[9] ***URL*** http://www.klab.com/jp/



照片 **6.2.1　DSAS** 的外观

本节将对 DSAS 的特征和内部结构进行介绍。

6.2.2　DSAS 的特征

首先我们来看一下 DSAS 的几大特征，然后再逐条进行讲解。

- 一个系统容纳多个网站
- 使用开源软件搭建

- ，网络服务都不会停止
- 服务器增设非常简单
- 故障修复非常简单

一个系统容纳多个网站

每当设立新的网站时都要重新搭建服务器或网络，此时整体拓扑结构就如图 6.2.1 所示。在这个拓扑中，当网站 A 的流量剧增时，即便此时的服务器已经无法进行这些处理，也不能挪用网站 B 的服务器来帮助其进行处理，因此就需要根据访问高峰时所需的服务器台数，为网站 A 增设服务器。若平时的访问量都比较多的话那还能接受，但如果只是为了暂时性的访问高峰（比如有优惠活动时）而增设服务器的话，成本就太大了。

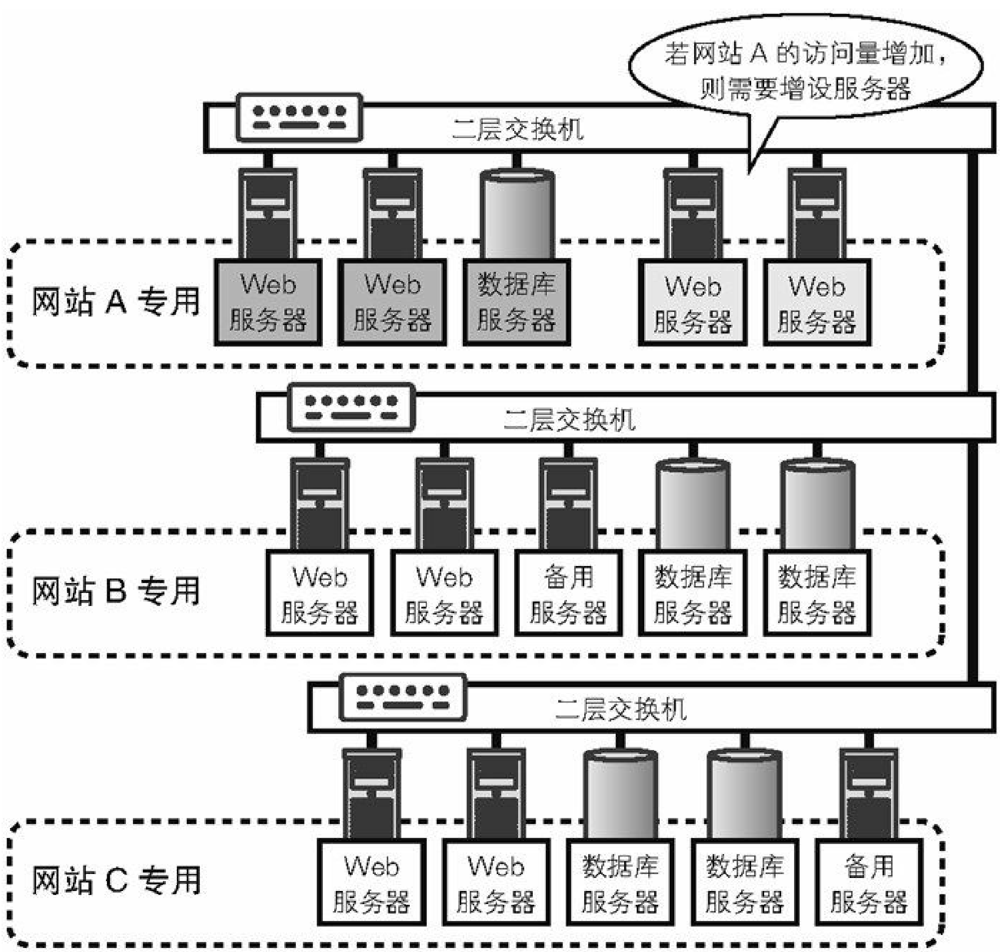


图 **6.2.1**　分别为各网站搭建系统

DSAS 的拓扑结构如图 6.2.2 所示，多个网站共用一个系统。而且网站使用的服务器也可以动态地进行改变 [10] 。如此一来，即使网站 A 的访问量剧增，也可以暂时利用网站 B 的服务器来应对。让一个系统容纳多个网站，可以避免服务器资源的浪费。

[10] “DSAS”的名字就是由“可以智能进行服务器的调配”这一特点得来的。

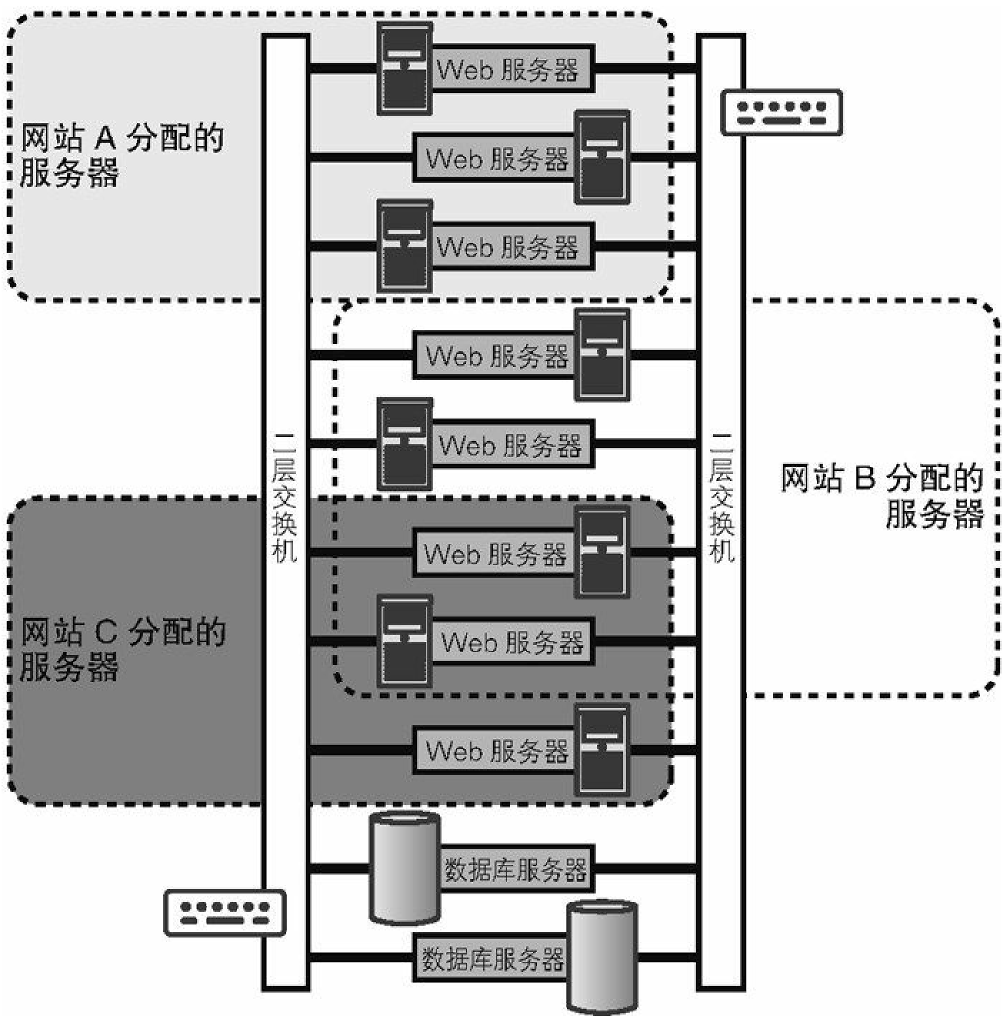


图 **6.2.2**　一个系统容纳多个网站

使用开源软件搭建

DSAS 的服务器架构如图 6.2.3 所示。虽然根据作用分为了“前端服务器”和“后端服务器”，但全部的服务器都是基于 Linux（Debian

GNU/Linux 4.0）这一开源系统搭建的。表 6.2.1 是使用的软件清单。关于前端服务器和后端服务器，请参考表 6.2.2 和表 6.2.3。

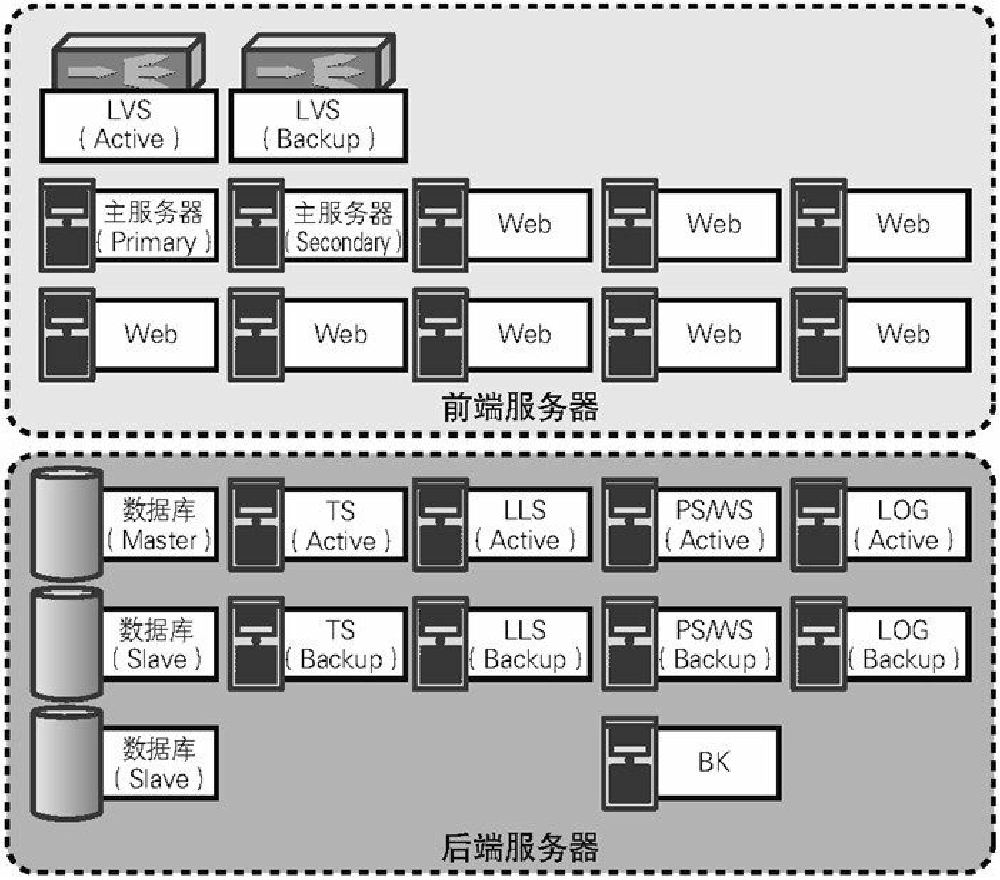


图 **6.2.3**　**DSAS** 的服务器架构

表 **6.2.1**　主要使用的软件

名称	版本
Apache	2.0版、2.2版
Tomcat	5.5版、6.0版
PHP	4.4版、5.2版

MySQL	4.0版、5.0版
qmail	1.03
djbdns	1.0.5
daemontool	0.70
thttpd	2.25b
dhcpd	3.0.4
atftpd	0.7
DRBD	0.7.25
stone	2.3c
keepalived	1.1.13
repcached	2.0

表 **6.2.2**　前端服务器

服务器
说明
LVS
通过使用Linux搭建的负载均衡器。可以将终端用户的请求分流到不同的Web服务器上进行处理。这里可以利用已经扩展了健康检查功能的keepalived，编写独自的维护脚本
主服务器
为了能够动态修改服务器的配置，必须保证“所有服务器的内容相同”。因此，在更

新网站时最好只对主服务器部署，其后再运行专用的命令扩展到所有服务器
Web
为终端用户提供Web服务的服务器。在所有的服务器上部署所有的网站，就可以让任何服务器为任何网站提供服务。Web服务器使用Apache，AP服务器使用Tomcat及PHP等

表 **6.2.3**　后端服务器

服务器
说明
数据库
数据库服务器使用MySQL。这是由一台Master和两台Slave构成的三台主机的最小架构，具体还可根据需要增加Slave的台数。当Master由于故障而停止时，将把Slave转换为Master进行修复
TS（Temporary Share）
保存临时数据（缓存或会话数据等）的数据库。使用了软件repcached（见下文），并在该软件上增加了同步到memcached的功能
PS（Permanent Share）、BK（Backup）
保存永久性数据（网站内容数据等）的存储服务器。利用第3章介绍的DRBD技术实现了冗余。这里不仅可以应用于NFS，而且在HTTP上也可以实现读取文件的操作。BK是备份服务器，定期备份PS服务器的文件
LLS
内部负载均衡器。与LVS一样使用keepalived搭建，被用于DNS或MySQL的负载分流等
LOG
收集Apache日志等的服务器。收集各Web服务器的日志并合并，被用于日志解析等。此外还使用了5.2节中介绍的Ganglia，来维持各服务器的正常运行

无论切断任何地方，网络服务都不会停止

图 6.2.4 是网络的物理接线图。因为完全实现了冗余，所以切断任何地方网络都不会中断。这里对二层交换机使用了 RSTP 进行冗余，服务器方面则使用了驱动绑定。Internet 线路有两条，即使一条链路失效，也没有什么影响。

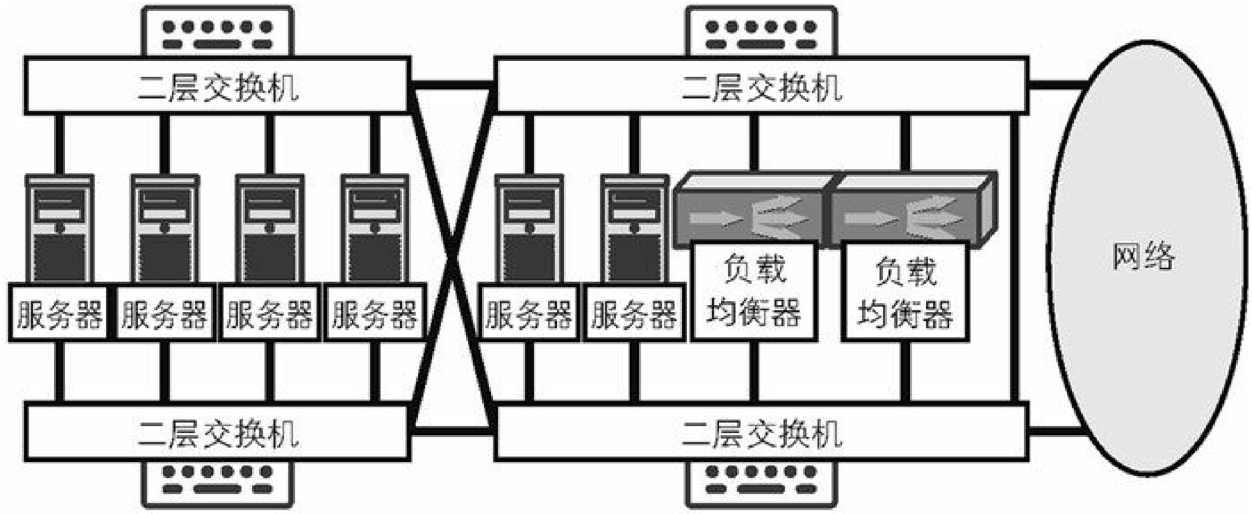


图 **6.2.4**　物理接线图

服务器增设非常方便

主服务器以外的服务器，全部实现了网络引导。网络引导的优点是增设服务器时不需要繁琐的部署工作。即使是刚买的磁盘崭新的服务器，只要在 BIOS 中开启网络引导，就能立即启动操作系统。根据启动时所输入的参数，可以让它成为 Web 服务器、负载均衡器或数据库服务器。

故障修复非常简单

使用 RSTP 进行二层交换机的冗余的情况并不少见，不过，图 6.2.4 的特点是 Internet 线路与二层交换机是直接连接的。负载均衡器是 Linux 机器，行为与其他服务器类似。因此，若 Internet 线路直接连接负载均衡器，必须在两台服务器（负载均衡器）上增加网卡。当负载均衡器由于故障等停止时，还需要为备份机器增加网卡并重新连接网线。

为了解决这个麻烦，这里让二层交换机来容纳 Internet 线路，将所有服务器的物理线路汇集在一起（详情见 3.3 节）。当负载均衡器出现故障时，选定一台服务器作为替代机并重新启动。此时，通过网络引导的启动参数指示其为负载均衡器。这项工作全部可以远程进行，即使不到数据中心也能实现故障的修复工作，实现了彻底的冗余结构。

6.2.3　系统架构的详情

DSAS 是由各种开源软件构成的。这里将选取 DSAS 的几个特点，对其

中精心设计的地方以及容易出问题的地方加以介绍。

使用驱动绑定的原因

当以冗余为目的使用多块网卡时，可能需要考虑为各个网卡分配 IP 地址，但是结果未必会像想象中的那样。例如，假设为某服务器的 eth0 分配了 192.168.0.1/24，为 eth1 分配了 192.168.0.2/24。这时如果从其他机器给这些地址发送 `ping`，这些地址都能正常返回应答。但是，如果拔掉 eth0 的 LAN 网线，即使 eth1 的 LAN 网线连接正常，也无法与 192.168.0.2 通信了。确认一下 ARP 表就会得知这些地址都被分配了 eth0 的 MAC 地址。也就是说，即使实际使用了两块网卡，实际上也只有一块网卡可以进行通信。

分配了多个 IP 地址的服务器路由选择表（Routing Table）如图 6.2.5 所示。像这样在对同一网络使用多个入口的情况下，基于路由选择表上的规则，发往 192.168.0.0/24 的数据包必须从 eth0 发出。这种行为是不考虑 网卡链路状态的，即使未连接 LAN 网线，也会尝试从 eth0 发送数据包。

```
# route -n
Kernel IP routing table
Destination Gateway Genmask         Flags Metric Ref   Use Iface
192.168.0.0 0.0.0.0 255.255.255.0   U     0      0       0 eth0  ←肯定使用
192.168.0.0 0.0.0.0 255.255.255.0   U     0      0       0 eth1  ←不会使用
```

图 **6.2.5**　使用多块网卡时的路由选择表

如果关闭发往 eth0 的路由选择表，虽然可以使用 eth1 与其他服务器通信，但此时必须清空 ARP 缓存列表，或者从 eth1 发送 Gratuitous ARP 通知 MAC 地址的变更。因此在网卡的冗余处理中，必须进行以下处理：

- 确认网卡的链路状态
- 如果链路失效，则切换网卡
- 发送 **Gratuitous ARP**

驱动绑定实现了这些功能。如果在虚拟接口上（bond0 等）注册的物理接口（eth0、eth1 等）发生链路故障，就会自动切换到链路正常的网卡并发送 GratuitousARP。这个功能对接入到二层交换机的冗余网络来说非常方便。

DRBP 实现故障转移时的注意事项

存储服务器使用 DRBD 进行了冗余，但是在发生故障进行故障转移时有几点需要注意。DRBD 中有“on-io-error”这个配置项目。在硬盘和 RAID 控制器出现故障导致物理设备的访问出错时，该项目会指定接下来进行什么样的处理，具体可以设定以下数值。

- **pass_on**：通知上层（文件系统）磁盘错误并继续运行
- **panic**：内核崩溃（**Kernel Panic**）
- **detach**：分离物理设备并继续运行

默认设定为 detach。笔者也基于下列原因选择了 detach。

- 一旦发生磁盘错误就造成内核崩溃会让人感觉过于极端
- 发生磁盘错误时，故障转移是最为理想的行为
- 设置为 **pass_on** 的情况下，除了文件访问失败的进程以外，其他异常都无法检测出来
- 设置为 **detach** 的情况下，因为能够立即监控出异常，所以能够顺利进行故障转移
- 设置为 **detach** 的情况下，操作系统能够照常运作，调查故障明细时会较为轻松
- 因为默认为 **detach**，所以不修改直接使用肯定也没有问题

当主服务器 Primary 遭遇磁盘错误时，最好能进行如下处理：

❶ 主服务器 **Primary** 隔离磁盘（操作系统继续运行）

❷ 实施故障转移的操作

❸ 将备用服务器 **Secondary** 升格为主服务器 **Primary**

但实际上往往不是这样。一旦物理磁盘遭遇故障，主服务器 Primary 会隔离磁盘继续运行，但备用服务器 Secondary 则会发生内核崩溃而终止运作。此时，各个服务器的内核日志中会输出如图 6.2.6 所示的内容。

```
●主服务器Primary的日志
kernel: drbd1: Local IO failed. Detaching...
kernel: drbd1: Sending NegDReply. I guess it gets messy.
kernel: drbd1: Notified peer that my disk is broken.

●备用服务器Secondary的日志
kernel: drbd1: Got NegRSDReply. WE ARE LOST. We lost our up-to-date disk.
kernel: Kernel panic - not syncing: drbd1: Got NegRSDReply. WE ARE
LOST. We lost our up-to-date disk.
```

图 **6.2.6**　磁盘故障时的内核日志

如果主服务器 Primary 对设备进行了隔离操作，就会向备用服务器 Secondary 发送 NegDReply 信息。备用服务器 Secondary 在收到 NegDReply 信息后，就会发生内核崩溃并终止运作。这里的源代码如代码清单 6.2.1 所示 [11] 。

> [11] 这是 drbd-0.7.25 软件的源码。

代码清单 **6.2.1**　**drbd/drbd_receiver.c**

```
STATIC int got_NegRSDReply(drbd_dev *mdev, Drbd_Header* h)
{
  sector_t sector;
  Drbd_BlockAck_Packet *p = (Drbd_BlockAck_Packet*)h;

  sector = be64_to_cpu(p->sector);
  D_ASSERT(p->block_id == ID_SYNCER);

  drbd_rs_complete_io(mdev,sector);
```

```
  drbd_panic("Got NegRSDReply. WE ARE LOST. We lost our up-to-date disk.\n"

  // THINK do we have other options, but panic?
  //       what about bio_endio, in case we don't panic ??

  return TRUE;
}
```

至少这是按照开发者的意图运行的，所以并不能说有什么问题。恐怕开发者是出于“即使停止也要保护数据”的意图进行实施的，但是如果让遭遇故障的服务器 Primary 继续运行，而让备用服务器 Secondary 终止运行的话，就不能进行故障转移的操作。因此，这里将 on-io-error 指定为 panic，让出现故障的服务器终止运行（磁盘遭遇错误时中断连接）。

配置 **SSL** 加速器

在使用 HTTPS 的网站中，如果在 Web 服务器上处理 SSL 请求，则会有性能降低的风险。虽然也可以如图 6.2.7 那样使用硬件加速器来减轻服务器的负载，但是在 DSAS 中，使用的是 **stone** [12] 这款 SSL 的软件加速器。

[12] ***URL*** http://www.gcd.org/sengoku/stone/Welcome.en.html

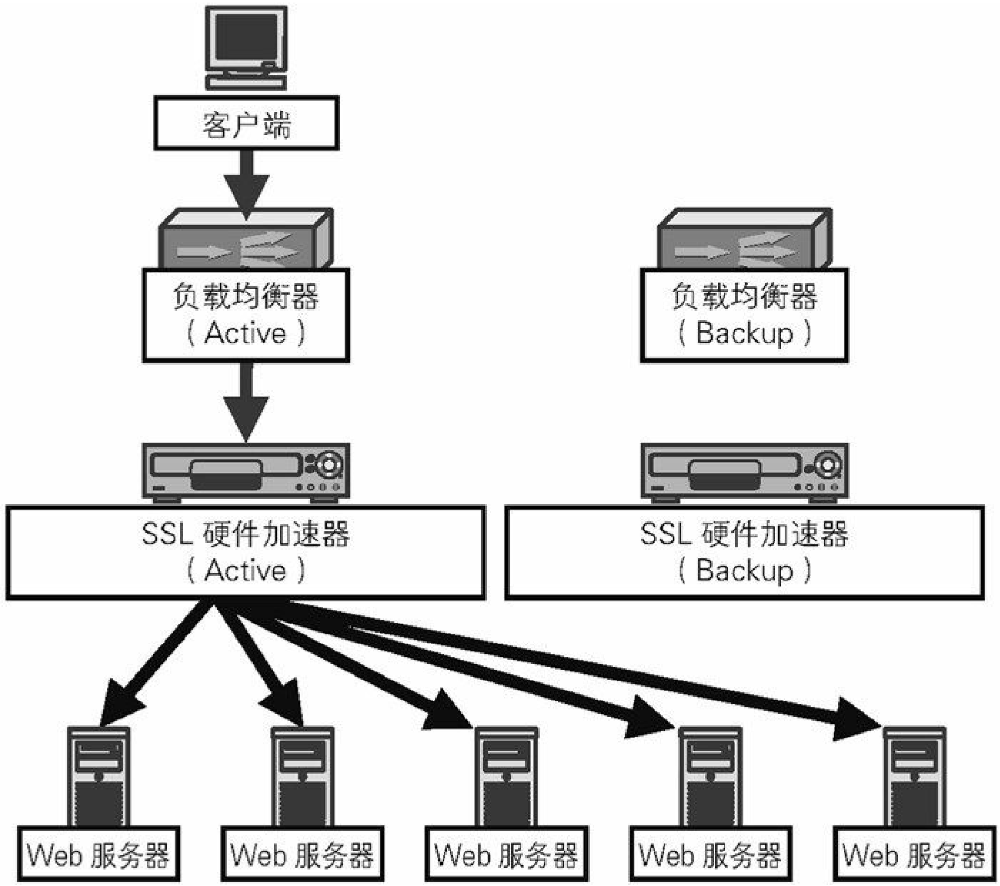


图 **6.2.7**　硬件加速器

硬件加速器多为比较昂贵的产品，但是为了能够快速地进行大量 SSL 的处理，一般使用经过冗余的两台设备进行处理工作。还有一些其他类型的产品，如带有加速器功能的网卡或 PCI 卡等，直接插在 Web 服务器上即可减轻 CPU 负载。但在使用 stone 这款软件加速器时，鉴于单台服务器处理能力较低，所以会像图 6.2.8 那样，将请求分流到多台服务器上进行处理。

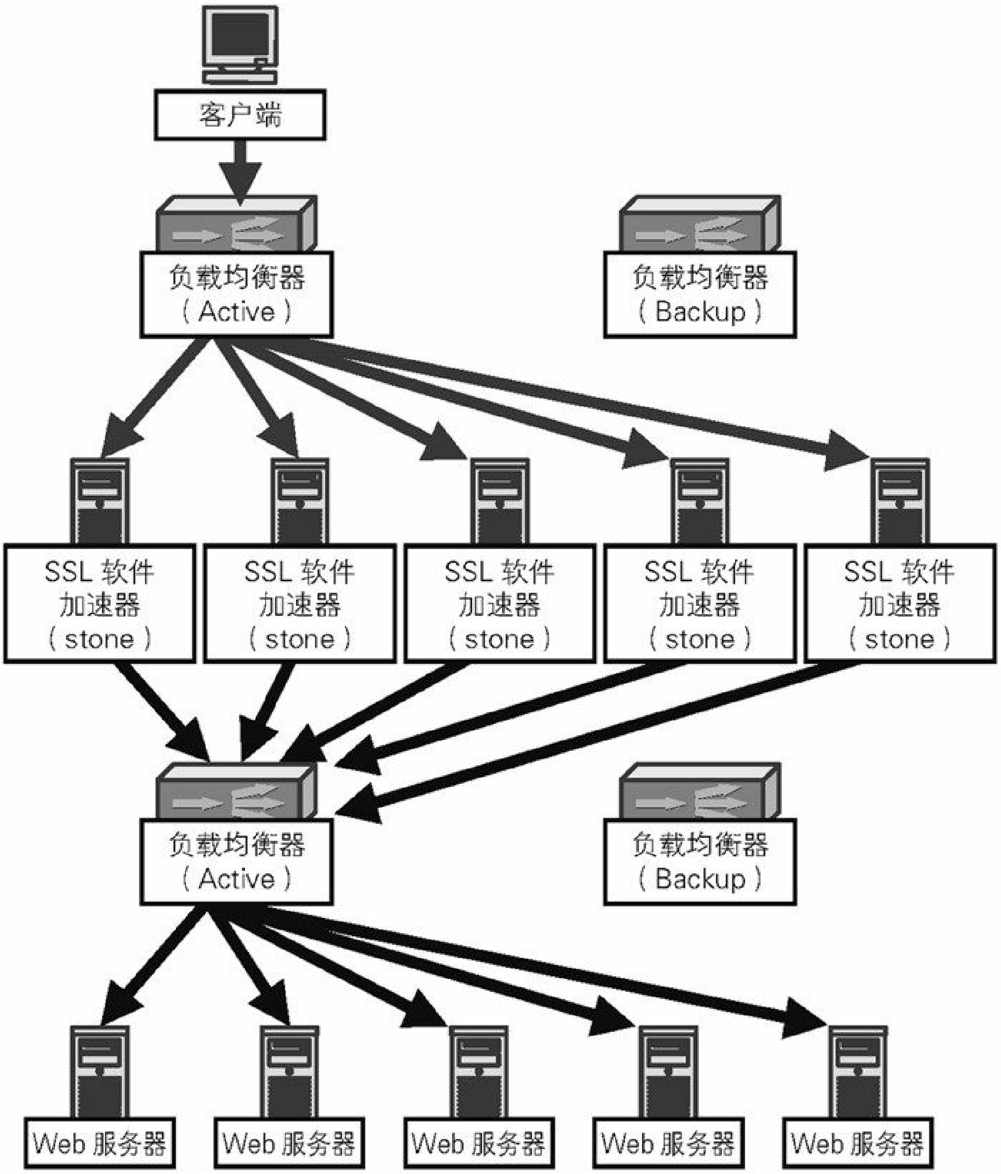


图 **6.2.8**　软件加速器

stone 在处理 HTTPS 连接时，将复合的 HTTP 请求经由负载均衡器传递给了 Web 服务器。因此在该结构中进行转接时，源 IP 地址会成为加速器的 IP 地址，所以某些 IP 地址的情况下就可能会遇到访问受到限制的问题。因此，需要将 stone 的配置改为像代码清单 6.2.2 那样，并将 客户端 IP 地址嵌入到 HTTP 消息头的“X-Orig-Client:”中。Web 服务器通过查看这个消息头就能知道实际上是从何处发出请求的了。

代码 **6.2.2**　**stone** 的配置

```
-z default
-z sid_ctx='ssl.example.com:443'
-z CApath=/usr/local/etc/ssl/certs/
-z cert=/usr/local/etc/ssl/cert.pem
-z key=/usr/local/etc/ssl/priv.pem
lls:80/proxy 443/ssl 'X-Orig-Client: \a' --
```

在 Web 应用程序中，有时还需要辨别客户端是通过 HTTP 连接的还是通过 HTTPS 连接的。如果是通过 HTTPS 连接的，则 Apache 的 mod_ssl 中会将环境变量设置为`HTTPS=on`。PHP、Tomcat 等 AP 服务器将根据这个环境变量进行判断，但是如果请求经由 stone 的话，全部连接都会变成 HTTP 连接，所以这里在配置文件 httpd.conf 中添加如代码清单 6.2.3 所示的设定。

代码清单 **6.2.3　Apache** 的配置

```
<IfModule mod_setenvif.c>
  SetEnvif Remote_Addr   "^192\.168\.1\." HTTPS=on
  SetEnvif X-Orig-Client "^$" !HTTPS
</IfModule>
```

Remote_Addr 会与运行有 stone 的服务器的地址进行匹配，同时设定`HTTPS=on`。接下来，查看 X-Orig-Client，若此处为空就表示不经由 stone，然后清除环境变量。这样一来，就只限在经由 stone 连接的情况下，才会进行`HTTPS=on`的设定。

扩展健康检查功能

由于在 DSAS 的负载均衡器中使用了 keepalived 工具，因此可以通过添加 keepalived 的补丁（keepalived-extcheck）[13] 对健康检查功能进行扩展。

> [13] 关于 keepalived-extcheck，请参考如下链接（日文）。
> ***URL*** http://d.hatena.ne.jp/hirose31/20090929/1254154720

通过应用此 keepalived-extcheck，就可以进行如下健康检查。

- **FTP_CHECK：** 检查 **FTP** 服务器是否能应答 **NOOP** 命令
- **DNS_CHECK：** 检查 **DNS** 服务器是否能返回响应
- **SSL_HELLO：** 检查服务器是否能应答 **SSL** 握手

keepalived 中还添加了 MISC_CHECK 功能，据此来进行一些原本不支持的健康检查。具体来说就是调用外部命令，由结束代码取得健康检查的结果，但这里又会出现以下问题：

- 鉴于每次健康检查都需要启动相关命令，因此消耗的性能成本会比较大
- 如果命令没能自己终止，进程会不断增加并累积得越来越多

关于 DNS 和 FTP 的健康检查，可以在进行 MISC_CHECK 后做出如代码清单 6.2.4 那样的配置。但是如果健康检查的周期比较短，比如检查周期是几秒钟的时候，调用外部命令就相对比较困难。因此我们可以像代码清单 6.2.5 那样，修改 keepalived 本身。这样相较于 MISC_CHECK，可以大幅减少健康检查的性能成本。

代码清单 **6.2.4**　通过 **MISC_CHECK** 对 **DNS** 和 **FTP** 进行健康检查

```
real_server 192.168.1.1 53 {
  MISC_CHECK {
    misc_path "/usr/bin/dig +time=001 +tries=2 @192.168.1.1 localhost.local
    misc_timeout 5
  }
}
real_server 192.168.1.1 21 {
  MISC_CHECK {
    misc_path "echo -en 'NOOP\r\nQUIT\r\n' | nc -w 5 -n 192.168.1.1 21 | eg
    misc_timeout 5
  }
}
```

代码清单 **6.2.5**　通过扩展的功能对 **DNS** 和 **FTP** 进行健康检查

```
real_server 192.168.1.1 53 {
  DNS_CHECK {
    port    53
    timeout 5
    retry   3
    type    A
    name    localhost.localdomain
  }
}
real_server 192.168.1.1 21 {
  FTP_CHECK {
    host {
      connect_ip   192.168.1.1
      connect_port 21
    }
    connect_timeout    5
    retry              3
    delay_before_retry 5
  }
}
```

通过 keepalived 中的 SSL_GET 可以取得 HTTPS 网站的状态代码。但需要留意的是每次健康检查时都要进行加密通信，这样就产生了一个问题，那就是在使用 HTTPS 的时候，不能同时使用需要进行客户端认证的页面以及非 HTTPS 的协议（SMTPS 等）。

用补丁扩展的 SSL_HELLO 会尝试进行 SSL 握手来检查服务器是否发送证书。因为此处并不依赖于应用软件协议，所以也能适用于需要客户端认证的网站，以及 SSL 加速器的健康检查。另外，因为这里不进行加密通信，所以还具有不加重服务器负载的好处。DSAS 使用了 SSL_HELLO 进行 stone 的健康检查。

既方便又安全地使用负载均衡器

负载均衡器的维护工作的主要是“配置虚拟服务器”和“分配真实服务器”两种。这项工作实际上就是修改配置文件（keepalived.conf）。但是当网站或服务器数量不断增多时，将变得很难直接编辑 [14] 。因此就需

要建立能够方便且安全地维护配置文件的机制。

> [14] 现在 DSAS 的 keepalived.conf 有 2500 行以上。

首先，忽略 keepalived.conf 的语法，只考虑维护时需要什么接口。维护负载均衡器的目的其实就是在真实服务器上对虚拟服务器进行增加、缩减或改变分配资源等工作。为了恰当地进行分配管理，必须能够直观地掌握目前具体哪个网站分配到了哪一个真实服务器。为此，这里就写了如下的配置文件。最左边的 w101、w102 是真实服务器的主机名称，SiteA、SiteB 是网站名称（虚拟服务器）。这个文件用 Klab 的术语来说就叫作“MATRIX”。该名字的由来是“因为像数学中的矩阵模型”。

```
w101: SiteA
w102: SiteA SiteC
w103: SiteB SiteC
w104: SiteB SiteC
w105: SiteB
w106
```

格式看起来很简单，但这里的格式到底是“网站名称：服务器 服务器”好呢？还是“服务器：网站名 网站名”好呢？笔者着实纠结了一段时间。如果是要掌握“给哪一个网站分配了哪一个服务器”，显然前者更容易理解。但是，实际上大多情况下都是在服务器出现故障导致配给到网站的资源失衡，或者移至其他服务器时编辑 MATRIX 的。例如当 w103 出现故障，需要切换至备用服务器 w106 时，如果这是后者的情况，因为只需调换 w103 及 w106 行，所以只进行一次剪切 & 粘贴即可；但如果是前者的话，就必须将字符串“w103”置换为“w106”才行。因此，在 MATRIX 中表示“哪个服务器为哪个网站提供支撑”时采用了后者的格式。

之后貌似只需参照 MATRIX 的格式来编写生成 keepalived.Conf 的脚本就可以了，但是因为 MATRIX 中不包括有关虚拟服务器的信息，所以除此之外还需要写一个如代码清单 6.2.6 那样的定义文件。这里使用了 YAML 格式来记录配置虚拟服务器所需的参数。DSAS 通过自身的 Perl 脚本读取 MATRIX 和这个定义文件，其后使用 Template-Toolkit 就可以生成 keepalived.conf 了。代码清单 6.2.7 是 Template-Toolkit 的模板。

代码清单 **6.2.6**　虚拟服务器的定义

```
PROJECT: SiteA
SERVICE:
  - 10.0.0.1:80

lb_algo:     lc
lb_kind:     DR
protocol:    TCP
HEALTH_TYPE: HTTP_GET
HTTP_GET:
  path:            /health.html
  status_code:     200
  connect_port:    80
  connect_timeout: 5
```

代码清单 **6.2.7**　**keepalived.conf** 的模板

```
virtual_server_group [% PROJECT %] {
[% FOREACH S=SERVICE -%]
  [% S.replace(':',' ') %]
[% END -%]
}
virtual_server group [% PROJECT %] {
  lb_algo    [% lb_algo %]
  lvs_method [% lb_kind %]
  protocol   [% protocol %]
[% FOREACH R=REAL -%]
  real_server [% R %] [% real_port %] {
    weight 1
    inhibit_on_failure
[% SWITCH HEALTH_TYPE -%]
[% CASE 'HTTP_GET' -%]
    HTTP_GET {
      url {
        path [% HTTP_GET.path %]
        status_code [% HTTP_GET.status_code %]
      }
      connect_port       [% HTTP_GET.connect_port    %]
      connect_timeout    [% HTTP_GET.connect_timeout %]
    }
[% CASE 'TCP_CHECK' -%]
    TCP_CHECK {
```

```
      connect_port [% TCP_CHECK.connect_port %]
      connect_timeout [% TCP_CHECK.connect_timeout %]
    }
[% END -%]
  }
[% END -%]
}
```

keepalived.conf 中必须根据所分配的服务器台数来描述 real_server 部分，这里使用模块引擎可以更加方便并安全地生成该文件。代码清单 6.2.8 就是自动生成的 keepalived.conf。

代码清单 **6.2.8**　自动生成的 **keepalived.con**

```
virtual_server_group SiteA {
  10.0.0.1:80
}
virtual_server group SiteA {
  lb_algo    lc
  lvs_method DR
  protocol   TCP
  real_server 192.168.0.1 80 {
    weight 1
    inhibit_on_failure
    HTTP_GET {
      url {
        path /health.html
        status_code 200
      }
      connect_port    80
      connect_timeout 5
    }
  }
  real_server 192.168.0.2 80 {
    weight 1
    inhibit_on_failure
    HTTP_GET {
      url {
        path /health.html
        status_code 200
      }
      connect_port    80
```

```
      connect_timeout 5
    }
  }
}
```

通过进行如上操作，在服务器需要进行分配调整时，只需编辑 MATRIX 并执行特定脚本就可以了，并不需要从庞大的 keepalived.conf 中找出变更的部分来手动修改配置。

处理会话数据

在负载分流环境中，鉴于用户在浏览不同页面时，连接到的未必是同一个服务器，因此不能在服务器本地文件里保存会话数据，而必须保存在类似数据库和 NFS 那样的云端上，这样才能保证 Web 服务器可以获取到相应的资源，但是由于数据库和 NFS 在访问集中时容易成为瓶颈，因此不适合用来保存频繁更新的会话数据。

关于会话数据的处理，下面将对“memcached”和“repcached”进行说明。

memcached

我们最初将高速缓存服务器的“memcached” 作为会话存储（Session Storage）来使用，但是因为 memcached 没有复制同步等功能，无法备份会话内容，因为一旦丢失会话数据（通常为会员登录或退出时的会话信息），就会为在线的会员带来困扰。

另外，会话数据一旦丢失，势必会对用户产生一些影响。例如，用户在进入下一页面时突然返回到主页，或者丢失所输入的数据等。为此，除了使用 memcached 以外，还在 NFS 服务器上使用 RamDisk 工具优化了 DRBD，这样就可以在重视性能的情况下选择 memcached，在重视安全性的情况下选择 RamDisk。类似上述这样，可以根据目前生产环境的情况来选择合适的会话存储方式。

但是，对 Web 应用程序来说，实现会话存储的切换会很麻烦，结果大多数情况下都只使用了通过 DRBD 冗余处理后的 RamDisk。虽说这里已经使用了 RamDisk 优化，但终究还是 NFS 服务器，为了消除已经过

期的会话数据，必须定期运行垃圾收集器（Garbage Collector）等，另外访问集中时还容易造成瓶颈的出现。

repcached

memcached 在性能和便利性方面非常出色。由于没有复制功能而无法使用，实在是有些可惜，于是便开发了为 memcached 添加复制功能 的 **repcached** [15] 。

> **15** 关于 repcached 的详情，请参考如下链接。
> ***URL*** http://book.51cto.com/art/201202/314930.htm

repcached 的运行情况如图 6.2.9 所示，两台服务器为一组，双向同步数据。无论在其中任何一个服务器中进行数据的设定，都能保存到双方的服务器中。

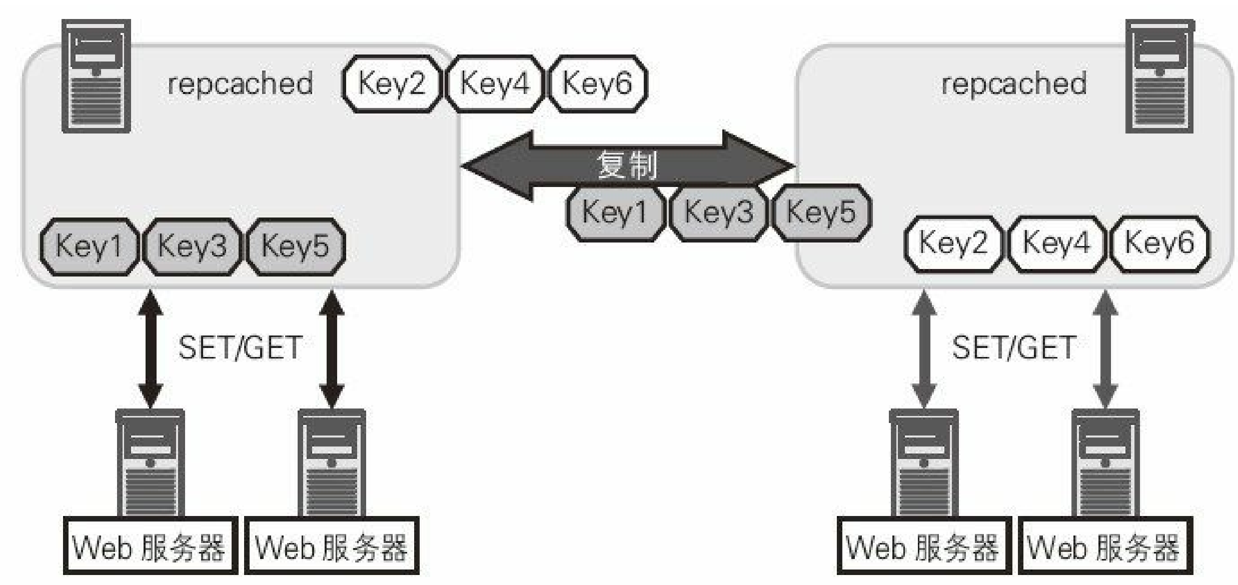


图 **6.2.9**　**repcached** 的运行情况

如图 6.2.10 所示，即使一边停止工作了，数据也能整个保存下来，所以 Web 服务器只需改变运行中的 repcached 连接，即可像什么事都没有发生一样继续处理。

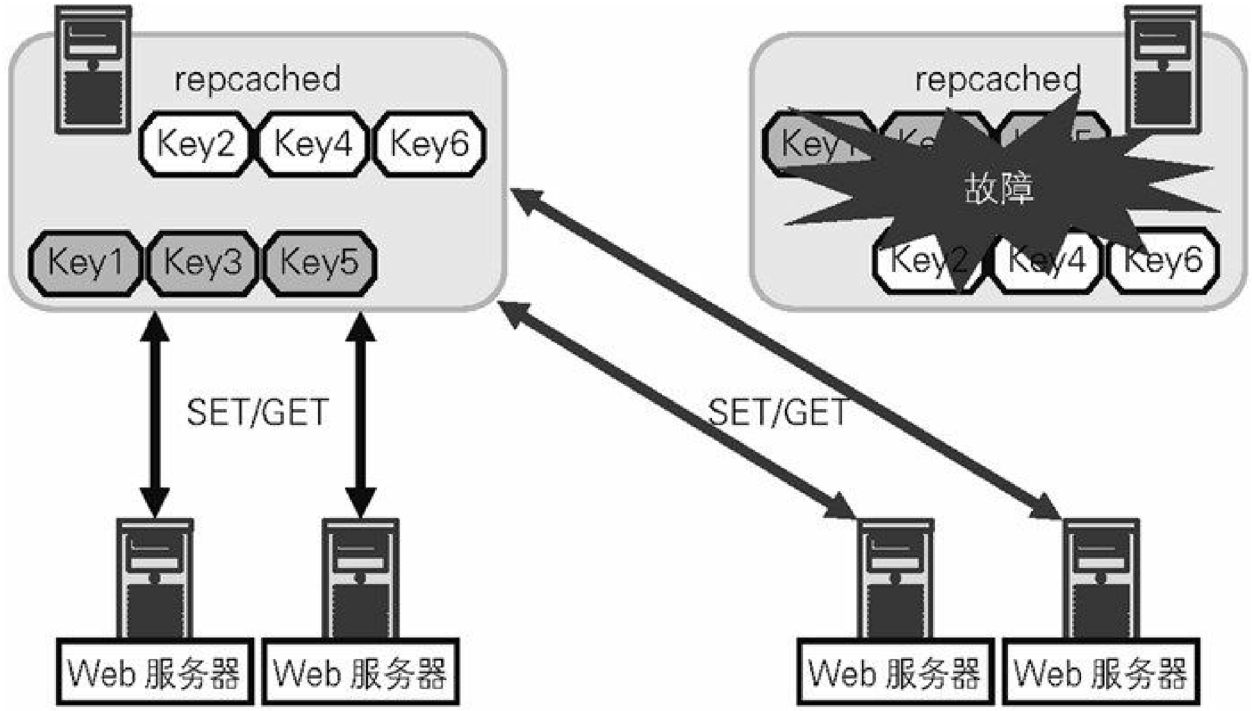


图 **6.2.10**　发生故障时的行为

运用 memcached 的服务器集群，可以向多个服务器进行负载分流，以及在发生故障时切换到备用服务器，所以即使不投入新的已经安装了 repcached 的服务器，安装有 memcached 的服务器也可以实现大部分的功能。

6.2.4　DSAS 的未来

DSAS 系统是以“根据情况智能地分配服务器资源”为目标命名的，但在目前的 DSAS 系统中，还尚未完全实现“智能地（Dynamic）分配服务器资源”。如果只闻其名而不知其原委，可能会对其有更好的印象，例如认为它是可以根据流量或连接数自动修改服务器架构的系统、服务器损坏时自动切换为备用机的系统等。

“Dynamic”有时还可能会被理解为“自动”的意思。确实，“Dynamic Routing Protocol”能够自动改写路由信息，“Dynamic DNS”能够自动配置 IP 地址。而且，DHCP 的首字母也是“Dynamic”，所以想必大家对“Dynamic”这个词都有个大致印象。DSAS 也将尽可能地不辜负当初的期望，努力做到名副其实。

根据流量的增加智能分配服务器资源是个很有趣的课题，而且当服务器出现故障时自动搭建其他服务器的冗余结构也不是说不可能实现。这样想着想着，慢慢地就可能有些异想天开了。但是不要总说这不可能，而是要认真考虑怎样才能实现，即使是半开玩笑也可以。在努力思考的过程中就会产生一些想法，而这些想法在某些状况下很可能就会发挥作用。虽然无法想象未来会如何发展，只希望能以“不断优化的智能系统”为目标而努力进步。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks
